Progressive Indexes
Timbó Holanda, P.T.

Citation
Timbó Holanda, P. T. (2021, September 21). *Progressive Indexes*. *SIKS Dissertation Series*. Retrieved from https://hdl.handle.net/1887/3212937

Version: Publisher's Version
License: Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from: https://hdl.handle.net/1887/3212937

Note: To cite this publication please use the final published version (if applicable).

Progressive Indexes

Pedro Holanda

Progressive Indexes

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van rector magnificus prof.dr.ir. H. Bijl,
volgens besluit van het college voor promoties
te verdedigen op dinsdag 21 september 2021
klokke 10:00 uur

door

Pedro Thiago Timbó Holanda
geboren te Fortaleza, Brazilië
in 1992

Promotiecommissie

Promotor:	prof. dr. Stefan Manegold	(CWI & Universiteit Leiden)
Copromotores:	dr. Hannes Mühleisen	(CWI & UvA)
	prof. dr. Peter Boncz	(CWI & VU)
Overige leden:	prof. dr. Aske Plaat	(Universiteit Leiden)
	prof. dr. Thomas Bäck	(Universiteit Leiden)
	prof. dr. Yanlei Diao	(École Polytechnique de Paris)
	dr. Stratos Idreos	(Harvard University)
	dr. Eduardo Cunha de Almeida	(UFPR)

The research reported in this thesis has been carried out within the Database Architectures group at Centrum Wiskunde & Informatica (CWI), the National Research Institute for Mathematics and Computer Science in the Netherlands.

SIKS Dissertation Series No. 2021-21 The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

This research is financially supported by the Dutch funding agency NWO, under project number 628.006.002 (the DAMIOSO project), in collaboration with Honda Research Institute, Leiden University and Centrum Wiskunde & Informatica (CWI).

Acknowledgments

I want to thank my direct supervisor, Stefan Manegold. I must say that I feel like I won the supervisor lottery. Stefan always gave me complete freedom to pursue any path I wanted during my studies while encouraging me to see the weak points of any ideas I had. To this day, I am still amazed by his ”eagle eyes” skill. His attention to detail is awe-inspiring, and having the opportunity of having him review all my papers, really enabled me to take them to the next level.

I want to acknowledge the impact that my co-supervisor, Hannes Mühleisen had on my thesis and post-PhD career. He provided me with the opportunity to work with him on DuckDB. It was one of the most fun projects I had during my Ph.D. and helped me improve my programming skills. In the final months of my Ph.D., when learning I had the intention to continue in the Netherlands for a few more years, Hannes also moved mountains to secure a post-doc position for me (on record time, I must say).

I have a high debt with Mark Raasveldt. I was fortunate to have Mark as a friend (and as a roommate for almost two years), and although we've been Ph.D. students at the same time, the reality is that Mark was already on another level. Living with him was an excellent opportunity to expand all the skill-set needed to be a successful researcher. Not only was he always available for any presentation/writing and programming questions I would have, but working with him was truly fun. With Mark, not all were related to work. As my flatmate, he also introduced me to all the greatness of Dutch culture (e.g., Febo, Action, New-Kids Turbo).

In the office, I was lucky enough to have the greatest officemate ever, Tim Gubner. Tim, thanks for all the great times and for singing with me all the greatest hits of Jon Lajoie, Tenacious D, and Backstreet Boys. I'm pleased that we could continue our musical journey even after being asked if we were killing rats in the office (although we were just singing slightly out of tune). I also think we still have the hidden potential of revolutionizing the computer science field with all of our great research ideas (e.g., Commie-coin, a communistic crypto-currency; DataBreaks: Breaks for Fast Databases). Thank you for all the laughs, and I hope we get to share an office again in the future!

During the Corona Pandemic, I was very thankful I managed to bring a little ”Brazilian Gang” into the Database Architectures group. Diego Tome and Matheus Nerone helped me maintain my sanity throughout the whole of 2020 by having daily coffee breaks, workouts, and video-game marathons. I think 2020 would be almost

unbearable without our activities. Besides all that, I also enjoyed working with Matheus on the first few months of the pandemic. I think we both did a great job motivating each other to finish off our multidimensional work, and I'm still impressed with what he accomplished with his mad plotting skills.

I would also like to thank my friends that made my years in Amsterdam some of the best years in my life. Especially Bianca Jabur (Kiki), Tijs Kramer, and George Anastasiou, thanks for all the fun!

Last but not least, I would like to thank my family for their support in all of my academic and life choices: my parents Tarcisio and Ana Holanda, and my sister Camila Holanda.

Contents

1 Introduction **11**

1 Data Analysis . . . 11

2 Interactive Data Analysis . . . 12

2.1 Index Creation Problem . . . 13

2.2 Research Questions . . . 15

3 Our Contributions . . . 16

4 Structure and Covered Publications . . . 16

2 Background **19**

1 Relational Database Systems . . . 19

1.1 Physical Layout . . . 20

2 Interactive Exploratory Data Analysis . . . 21

3 Index Structures . . . 22

4 Index Selection Problem . . . 23

4.1 Automatic Index Selection . . . 24

4.2 Adaptive Index Creation . . . 24

3 Progressive Indexing **27**

1 Introduction . . . 27

1.1 Contributions . . . 29

1.2 Outline . . . 29

2 Related Work . . . 29

2.1 Cracking Kernels . . . 30

2.2 Adaptive Indexing for Robustness 31
3 Progressive Indexing 35
3.1 Progressive Quicksort 37
3.2 Progressive Radixsort (MSD) 38
3.3 Progressive Bucktersort 40
3.4 Progressive Radixsort (LSD) 42
4 Greedy Progressive Indexing 43
4.1 Greedy Progressive Quicksort 44
4.2 Greedy Progressive Radixsort (MSD) 46
4.3 Greedy Progressive Bucketsort 47
4.4 Greedy Progressive Radixsort(LSD) 47
5 Experimental Analysis 48
5.1 Setup. 48
5.2 Delta Impact 50
5.3 Cost Model Validation 53
5.4 Interactivity Threshold 56
5.5 Varying Interactivity 59
5.6 Adaptive Indexing Comparison 61
6 Summary 66

4 Multidimensional Progressive Indexing 69

1 Introduction 69
1.1 Contributions 70
1.2 Outline 70
2 Related Work 71
2.1 Multidimensional Data Structures 71
2.2 Adaptive/Progressive Index 73
3 Multidimensional Progressive Indexing 75
3.1 Data Structure 76
3.2 Creation Phase 77
3.3 Refinement Phase 80
3.4 Greedy Progressive Indexing 82
4 Experimental Analysis 85
4.1 Setup. 85
4.2 Data Sets & Workloads 86
4.3 Delta Impact 87

4.4 Performance Comparison . . . 91
4.5 Impact of Dimensionality . . . 96
4.6 Full Scan Exceeding the Interactivity Threshold . . . 97
5 Summary . . . 98

5 Progressive Merges 101
1 Introduction . . . 101
1.1 Contributions . . . 102
1.2 Outline . . . 102
2 Related Work . . . 102
2.1 Merge Complete (MC) . . . 103
2.2 Merge Gradual (MG) . . . 104
2.3 Merge Ripple (MR) . . . 104
3 Progressive Mergesort . . . 106
4 Experimental Analysis . . . 110
4.1 Setup . . . 111
4.2 Performance Comparison . . . 112
4.3 Varying Data Sizes . . . 113
4.4 Appends during Index Creation . . . 115
5 Summary . . . 116

6 Big Picture 117
1 The Elephant In The Room . . . 117
2 Future Work . . . 118
2.1 Progressive Indexes . . . 118
2.2 Progressive Merges . . . 119

Summary 123

Samenvatting 125

Publications 127

CHAPTER 1

Introduction

1 Data Analysis

Data Analysis is the process where a scientist extracts valuable knowledge (e.g., data correlation, useful patterns, market trends) and uses this information to make decisions.

Using the car industry as an example, car engineers generate random deformations on car shapes to better understand how design changes affect aerodynamics. The actual process of analyzing the generated simulation data is done by data scientists [33]. For example, the data scientists might learn that a slightly different position of the car's rear-view mirror might significantly impact its aerodynamics. This process consists of generating random deformations to a car shape (i.e., slightly changing x-y-z positions of the car model) and running it through computational fluid dynamic simulations. Each simulation generates raw files consisting of many gigabytes of data. Due to the data size, the data scientist must apply interactive data analysis techniques to learn which modifications in the car's shape improve its aerodynamics.

One major problem with this approach is that each simulation takes many hours to be executed. One promising solution is to train a machine learning model to generate the simulated data when receiving a car shape as input. Such a model is trained over many gigabytes of already executed simulations and eliminates the necessity of running simulations for new deformations [20, 19]. This type of data analysis pipeline is relatively common in the industry and can be summarized into three main steps:

1. **Pre-Processing:** In this step, data is generated, cleaned, and loaded in a database management system (DBMS).
2. **Interactive Data Analysis:** In this step, the data scientist explores the data sets to gain insights about the data.
3. **Machine Learning Driven Analysis:** A machine learning model is created to accelerate data generation, classification, and prediction.

In this thesis, we focus on optimizations for Interactive Data Analysis. In the following sections of this chapter, we will introduce this topic and present the research questions explored in this thesis.

2 Interactive Data Analysis

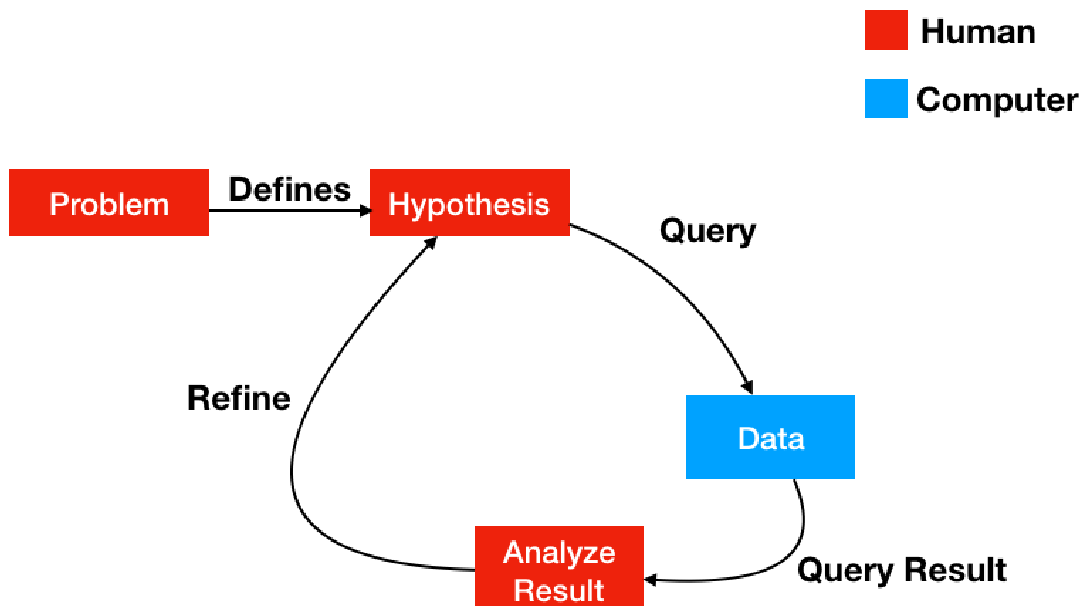


Figure 1-1: Interactive Data Analysis Workflow

Data scientists perform exploratory data analysis to discover unexpected patterns in large collections of data. This process is done with a hypothesis-driven trial-and-error approach [52]. Figure 1-1 depicts the classical interactive data analysis problem. The data scientists derive hypotheses and test them by querying segments that could potentially provide insights. With this result, they refine their original hypotheses and either zoom in on the same segment or move to a different one depending on the insights gained.

In this workflow, the data scientist is always in the loop and depends on fast query responses to perform interactive data analysis. The study by Liu et al. [41] shows that any delay larger than 500ms (the "interactivity threshold") significantly reduces the rate at which users make observations and generate hypotheses.

When dealing with small data sets, providing answers within this interactivity threshold is possible even when only performing full scans on the data. However, exploratory data analysis is often performed on larger data sets as well. For example, the SkyServer project [56] which maps the universe, consists of many terabytes of data. This project has many interactive queries, with data scientists checking different hypotheses on small segments of the sky. Due to the massive amount of data, answering these queries under the interactive threshold by performing only full scans is unfeasible.

An essential optimization that these highly selective queries require is the exploitation of secondary index structures. Depending on the query's selectivity, an index structure can diminish the execution time in orders of magnitude, allowing for responses in interactive times.

2.1 Index Creation Problem

Index creation is one of the major difficult decisions in database schema design [15]. Based on the expected workload, the database administrator (DBA) needs to decide whether creating a specific index is worth the overhead in creating and maintaining it. Creating indexes up-front is especially challenging in exploratory and interactive data analysis, where queries are not known in advance, workload patterns change frequently, and interactive responses are required. In these scenarios, data scientists load their data and immediately want to start querying it without waiting for index construction. Also, it is not certain whether or not creating an index is worth the investment at all. We cannot be sure that the column will be queried frequently enough for the large initial investment of creating a full index to pay off.

Despite these challenges, indexing remains crucial for improving database performance. When no indexes are present, even simple point and range selections require expensive full table scans. When these operations are performed on large data sets, indexes are essential to ensure interactive query response times. Two main strategies aim to release the DBA of having to choose which indexes to create manually. The first step at automatizing the index creation problem was self-tuning tools. These tools perform offline (i.e., indexes are created when the database is not being used) and online (i.e., indexes are created while queries are executed) full index creation.

The second step was adaptive indexing techniques which perform incremental index creation (i.e., indexes are created partially during query execution).

Self-Tuning Tools

Self-tuning tools [1, 14, 58, 23, 13, 11, 44, 53] are pieces of software that gives hints to the Database Administrator (DBA) on which indexes to create for a database. Those hints are an attempt to find the optimal set of indexes given a query workload. These tools consider the benefits of having an index versus the added costs of creating the entire index and maintaining it during modifications to the database.

Self-tuning tools are very successful in traditional OLAP/data warehouse scenarios (e.g., Producing reports). In these scenarios, there is a lot of workload knowledge, the workloads do not change regularly, and the systems are idle off-working times to perform full index creation.

However, these tools are not suitable for index creation in exploratory data analysis due to four main reasons. (1) They require a priori knowledge of the expected workloads. (2) They do not quickly adapt to frequently changing workloads. (3) They require idle time to perform full index creation. (4) The data scientist must take the database administrator's role in analyzing the hints produced by the tools to decide which indexes should be created ultimately.

Adaptive Indexing Techniques.

Adaptive indexing techniques such as database cracking [36, 21, 50, 49, 26, 35, 37, 47, 46, 25, 34, 29] are a more promising solution for the index creation problem in interactive data analysis. They focus on automatically and incrementally building an index as a side effect of querying the data. An index for a column is only initiated when it is first queried. As the column is queried more, the index is refined until it eventually approaches a full index's performance. In this way, the cost of creating an index is smeared out over the cost of querying the data many times, though not necessarily equally, and there is a smaller initial overhead for starting the index creation.

Although adaptive indexing techniques can alleviate the shortcomings of self-tuning tools, they introduce new issues that have not been completely tackled yet. (1) The first query cost can be much higher than a simple full scan cost. (2) There is no guarantee of robustness, and (3) There is no guarantee that the index will eventually converge to a full index (i.e., index all points in the dataset).

First Query Cost. When executing a query on a column for which no index

has been created yet, a full copy of the data is performed to a secondary index structure. After completing the copy, the data is then partitioned into one or more pieces depending on the used adaptive technique. This process incurs a much higher cost than simply scanning the data to answer the query.

Robustness. In general, adaptive indexing only refines pieces that are accessed. When the same piece is constantly requested, its access time becomes similar to a full index. However, as soon as the data scientist decides to query a less refined piece, the query performance degrades, causing performance spikes.

This scenario is highly undesirable for the user since it brings unpredictability to the query response time. Similar queries (i.e., queries that inspect the same amount of data in a column) can have widely different response times.

Convergence. In general, only accessed data points are added to the index structure. Although adaptive indexing can achieve near full index response time when pieces are sufficiently refined, it will not guarantee a full index response time to any filter predicates unless all data points were used as filter predicates.

2.2 Research Questions

Our research aims to investigate how indexes can be created and refined in a similar process as adaptive indexing while inflicting a low indexing penalty on the initial queries, enforcing a predictable query response time and with guaranteed full index convergence. Ultimately, we envision an indexing technique, called *Progressive Indexing* that mitigates these drawbacks from adaptive indexing by performing a more fine-grained refinement and progressively converging to a full index.

We define our main research question as follows:

Research Problem 1 How can we create/refine indexes during query execution with a low impact over initial queries, predictable query response time, and guaranteed full index convergence?

Research Problem 2 How can we create progressive indexes for queries with filters on multiple attributes?

Research Problem 3 How can we update progressive indexes while keeping predictable query performance and guaranteed full index convergence?

3 Our Contributions

This thesis describes how to create progressive indexing techniques for both unidimensional and multidimensional index structures. For each, we explore the current state-of-the-art on adaptive indexing and attempt to improve the characteristics defined in our main research question.

Its main contributions are as follows:

- **Progressive Indexing (Chapter 3).** We alter various sorting algorithms (i.e., Quicksort, Radixsort - Most Significant Digit, Radixsort - Least Significant Digit, and Bucketsort Equiheight) to work progressively following a pre-defined indexing budget.
- **Greedy Progressive Indexing (Chapter 3).** We define a cost model for each Progressive Indexing algorithm that allows for automatic selection of the indexing budget.
- **Progressive KD-Tree (Chapter 4).** We propose a multidimensional progressive indexing, the Progressive KD-Tree, that progressively builds KD-Trees for queries with filters in multiple dimensions.
- **Greedy Progressive KD-Tree (Chapter 4).** We define a cost model for our Progressive KD-Tree algorithm, allowing for an automatic selection of the indexing budget.
- **Progressive Merges (Chapter 5).** We define a new progressive indexing technique used to merge appends into progressive indexes.

4 Structure and Covered Publications

Chapter 3 describes the progressive indexing techniques for Quicksort, Radixsort - Most Significant Digit, Radixsort - Least Significant Digit, and Bucketsort Equiheight in their traditional and greedy formats. This chapter is based on the following papers:

- **Progressive Indexes: Indexing for Interactive Data Analysis [32].**
  Pedro Holanda, Mark Raasveldt, Stefan Manegold and Hannes Mühleisen
  46th International Conference on Very Large Databases (VLDB 2020)

- **Progressive Indices – Indexing Without Prejudice [28].** Pedro Holanda, 44th International Conference on Very Large Data Bases (VLDB 2018, PhD Workshop)

Chapter 4 presents both traditional and greedy versions of multidimensional progressive indexing, a technique based on quicksort and KD-Trees. This chapter is based on the following papers:

- **Multidimensional Adaptive & Progressive Indexes [43].**
  Matheus Nerone, Pedro Holanda, Eduardo Almeida and Stefan Manegold
  37th IEEE International Conference on Data Engineering (ICDE 2021)

- **Cracking KD-Tree: The First Multidimensional Adaptive Indexing [31].**
  Pedro Holanda, Matheus Nerone, Eduardo Almeida, and Stefan Manegold, 7th International Conference on Data Science, Technology and Applications (DATA 2018, EDDY)

Chapter 5 describes Progressive Merges, a technique developed to progressively merge batches of appends into progressive indexes without impacting the predictability of the queries. This chapter is based on the following paper:

- **Progressive Mergesort: Merging Batches of Appends into Progressive Indexes [30].** Pedro Holanda and Stefan Manegold, 24th International Conference on Extending Database Technology (EDBT 2021)

Finally, Chapter 6 summarizes the work done in progressive indexing, discusses its challenges, and provides future work.

CHAPTER 2

Background

In this chapter, we will cover the basic knowledge necessary to read this thesis. We will start with an overview of Relational Database Systems (Section 1) and their physical layout (Section 1.1). We continue with an overview of Interactive Exploratory Data Analysis (Section 2) and give a general explanation of indexing techniques and how they can be used to boost interactive exploratory queries (Section 3).

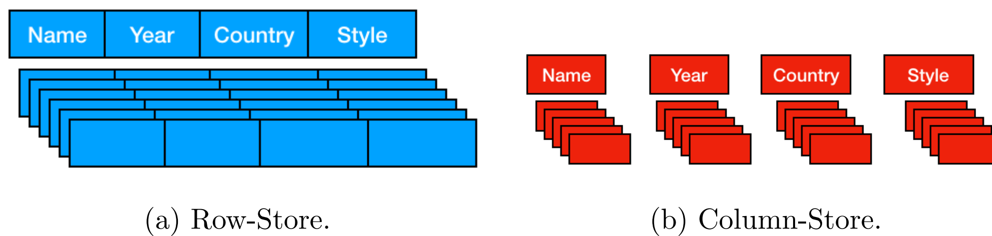
1 Relational Database Systems

Relational Database Systems (RDBMS) have been around since the early 70s. They are essential to any application that must access persistent data. They implement various techniques that guarantee data integrity, fast data access, transaction control and overall facilitate the development of a new application. The programmer does not need to worry about which data structures to represent his data, how to guarantee ACID (Atomicity, Consistency, Isolation, Durability) properties, or protecting his data against different types of corruption (e.g., hardware failures).

As an example, consider that a developer wants to create an online music store. He must store information about artists (e.g., their name, year they started, their country, and music style) and about their albums (e.g., album name, the year they were released, and the artist that made it). A simple way of storing this data would be to use text files (e.g., CSVs). However, the developer now has to implement methods to scan and write these files while being smart enough to store them on efficient data

structures. He also must use the correct algorithms to join the data in these files. And must deal with representation issues (e.g., how to store an album made as a collaboration of multiple artists?), transaction issues (e.g., what happens if two users alter the file simultaneously?) and data corruption (e.g., what happens if we are writing on the file and we experience a power shortage?). A more straightforward solution is to use RDBMSs, since they are designed to tackle these problems.

1.1 Physical Layout



(a) Row-Store. (b) Column-Store.

Figure 2-1: Physical layout of relational databases.

RDBMSs do not store data as text files but rather as a table composed of n columns, where every row of this table represents a different entity with values for each of these columns (See Figure 2-1). An essential physical layout decision is choosing how the data should be partitioned, and there are two primary ways of doing it, a row-store or a column-store.

In the row-store model, data is partitioned in rows (i.e., the rows are stored consecutively in memory). This model is preferred for transactional workloads (i.e., when most queries update only a few tuples) since individual rows can be fetched computationally cheap. This model's main disadvantage is when you must retrieve a lot of data but not from all columns. Since rows are stored consecutively in memory, you will fetch data from columns you are not interested in, essentially wasting time. In a typical analytical scenario, the user is only interested in a small set of columns from the table, making this format unfit for data analysis.

In the column-store model, data is partitioned per column (i.e., the columns are stored consecutively in memory). This model is preferred for analytical queries since it is cheap to fetch individual columns, resulting in immense savings on disk access and memory bandwidth.

As an example, suppose our music-store table has 100 gigabytes of data, and different from the Figure 2-1 it is composed of 100 columns, also assume that the columns occupy the same amount of storage, 1 gigabyte per column. When executing an

analytical query interested in the number of albums released in 1980, the performance would significantly differ depending on the layout. In a row store, reading one column is equivalent to fetching all tuples, which at 100 megabytes per second (i.e., a typical hard-disk transfer speed) would take us about 17 minutes. In a column store, the same query can fetch the column that stores the albums' release date separately, so we only need to read 1 gigabyte of data, which takes about 10 seconds.

2 Interactive Exploratory Data Analysis

The workload from interactive data analysis is a type of analytical workload. The data scientist inspects a massive amount of data by issuing selective analytical queries (sometimes via a visualization tool) to test their hypothesis.

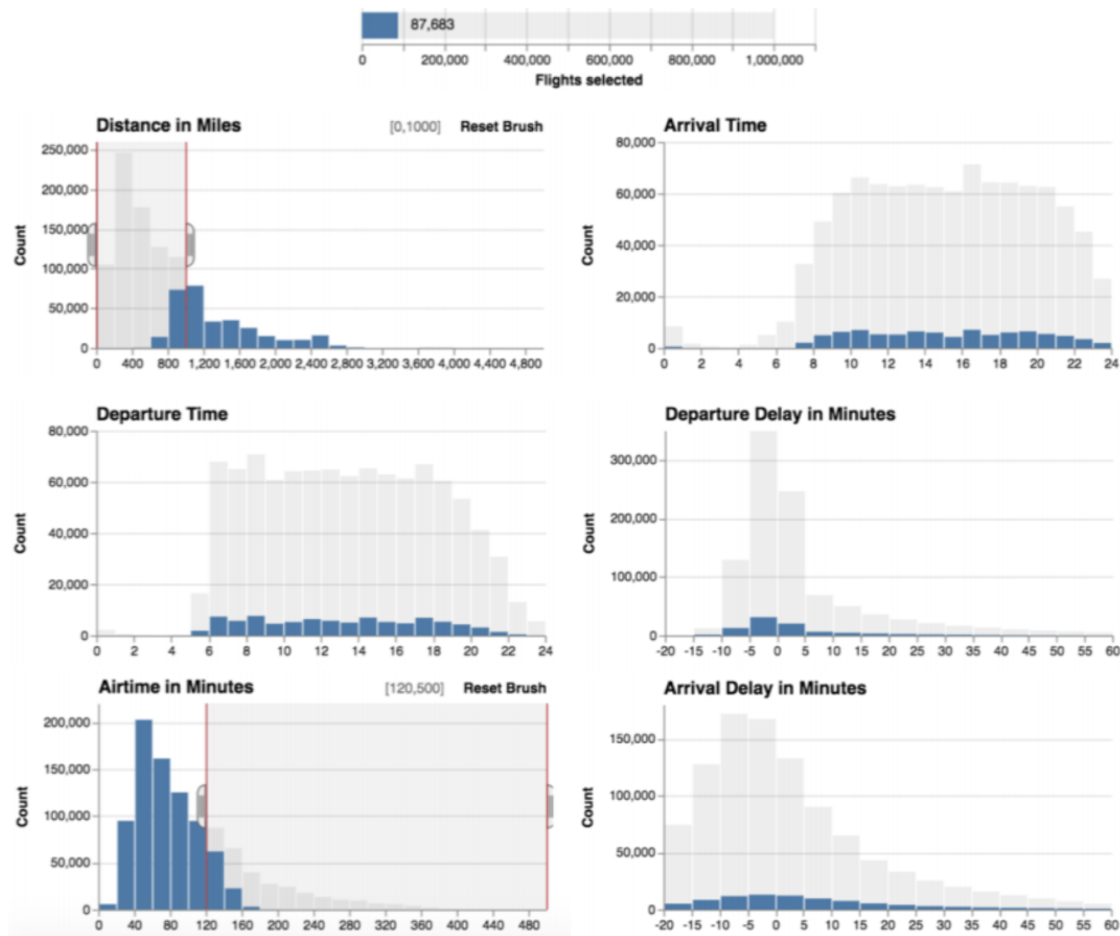


Figure 2-2: Interactive Data Analysis Example [4]

In [4], Battle et al. present cross-filter applications as the classical scenario of interactive data analysis. These applications consist of aggregate-filter-group queries

with users expecting almost immediate responses from the system.

Figure 2-2 depicts an example of a cross-filter application. It presents a dataset that contains flight information with six different attributes. The idea is that the data scientist can visualize each attribute as one of the histogram figures (e.g., the distance in miles histogram presents, from our selected flight, the number of flights that traveled a given amount of miles). The data scientist must interact with the range slider on top, and these figures are automatically updated depending on the filter's new inputs. It is easy to imagine that it will be quite frustrating if these figures are not immediately updated when changing the filter.

Since these workloads are dependent on a filter, when applying selective filters (e.g., wanting to know the information of a small number of flights), aggressive data skipping techniques like secondary index structures can significantly influence the query performance.

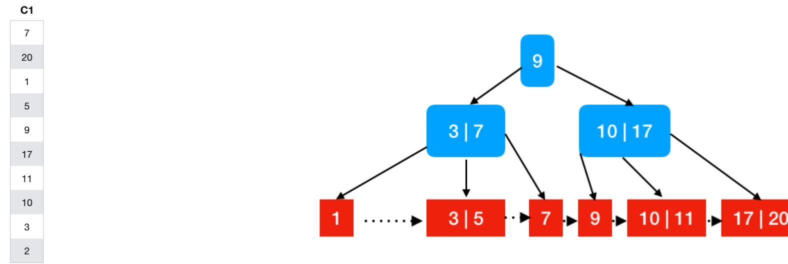
Let's go back to our music-store example from section 1.1, and let's assume that we want to know the quantity of all albums released in 1980 (also assuming that the selectivity is around 0.1% of all our data). When no index is present, a full scan of the column must be performed, which takes approximately 10 seconds. When using an index, we can access just the data that match our filter. Hence we only scan 0.1% of our data, with our query taking around 0.01 seconds to be fetched.

3 Index Structures

From our previous example, it becomes clear that, for highly selective queries (i.e., queries that filter most of the data), an index structure can significantly impact query performance. This impact exists because index structures can skip data that is not relevant to our query (i.e., not reading data that does not match our filter predicates).

Index structures come in all shapes and forms, covering different use cases. For example, the Adaptive Radix Tree (ART) [2, 40, 8] is designed to produce a compact index structure that is efficient for point-queries (i.e., queries with equality filters) and updates. At the same time, the B+Tree [22, 60] is optimized to execute range queries while not being as efficient as the ART for point-queries and updates.

Figure 2-3a depicts an example where the original data is composed of one column with unordered integers, and Figure 2-3b depicts a B+tree index created on this column. Note that the B+tree has the original data sorted in its leaves (i.e., red nodes) while the inner-nodes (i.e., blue nodes) are used to navigate the tree efficiently. When executing the following query `SELECT SUM(R.C1) FROM R WHERE R.C1 BETWEEN 3`



(a) Original Column (Column-Store). (b) B+-Tree Index.

Figure 2-3: Scan Vs Index.

AND 6, if we do not have an index, that means we must scan all the elements from our original column. However, if a B+Tree exists, we can quickly navigate the inner nodes and scan only the leaves with relevant data.

4 Index Selection Problem

A natural question arises after understanding the benefits of indexes. Why not create all possible indexes to speed up all possible filter queries? Although indexes boost query execution since they skip data that does not match filter predicates, they impost three different penalties to the DBMS. Indexes have a creation cost, a maintenance cost (i.e., every time an update happens, the index must be updated as well), and a storage cost (i.e., secondary indexes materialize a copy of the original data). Hence, the DBA must decide which indexes to create for a given database.

The DBA's goal is to decide a set of indexes to create for a table that will execute the workload as fast as possible while considering the amount of available memory. To do so, the database administrator must follow four steps: (1) Identify a relevant workload, (2) Create a search space with indexes that can potentially speed up this workload, (3) Perform a careful analysis on the maintenance and speed up trade-offs, (4) Assess the impact on the available memory.

Even when workloads are well known, selecting the optimal set of indexes is an NP-Hard problem [15], since it represents an analysis on all possible combinations of indexes that can be helpful to the workload. When the querying pattern is not known in advance, optimal a-priori index creation is impossible. To facilitate this process, two different types of solutions have been proposed. (1) automatic index selection and (2) adaptive index creation.

4.1 Automatic Index Selection

Automatic index selection techniques [1, 14, 58, 23, 13, 44, 53, 11] attempt to automatize the index selection process either completely or by giving hints of what indexes to create or drop to the DBA. In general, they work by capturing the workload, finding a set of indexes that optimize it, and either suggesting them for the DBA to create or by automatically creating them.

The process of finding a set of indexes can be driven by machine learning algorithms [44], or by the what-if architecture [13]. In the what-if architecture, the DBMS' query optimizer is used to predict the workload boost and the extra costs of maintaining and creating indexes using hypothetical indexes (i.e., it only creates the index's meta-data to force the optimizer to predict the costs if the index existed).

These solutions are well suited for the classical data warehouse scenario since the data warehouse scenario has a well-defined workload that rarely changes and has maintenance times (i.e., hours when the database is not being queried). The DBMS can exploit the maintenance time to perform full index creation. Since self-tuning tools can only be used when the system's workload is stable and known, they present several problems for interactive data analysis workloads. In an interactive environment, the workload is unknown or rapidly changes beyond what is known upfront. Besides, there is no specific idle time to invest in upfront full index creation. Hence automatic index selection techniques do not offer much help.

4.2 Adaptive Index Creation

Adaptive indexing techniques are an alternative to a-priori index creation. Instead of constructing the index upfront, the index is built as a by-product of querying the data. These techniques are designed for scenarios where the workload is unknown, and there is no idle time to invest in index creation. Their main goal is to smear out the high investment of creating an up-front full index over the execution of several queries.

Database Cracking [36] (also known as "Standard Cracking") is the original adaptive indexing technique. It works by physically reordering the index while processing queries. It consists of two data structures: a cracker column (i.e., a copy of the original column) and a cracker index (i.e., a binary search tree that holds information on where pieces offsets and maximum value).

Each incoming query cracks the column into smaller pieces and then updates the cracker index concerning those pieces. As more queries are processed, the cracker index converges towards a full index.

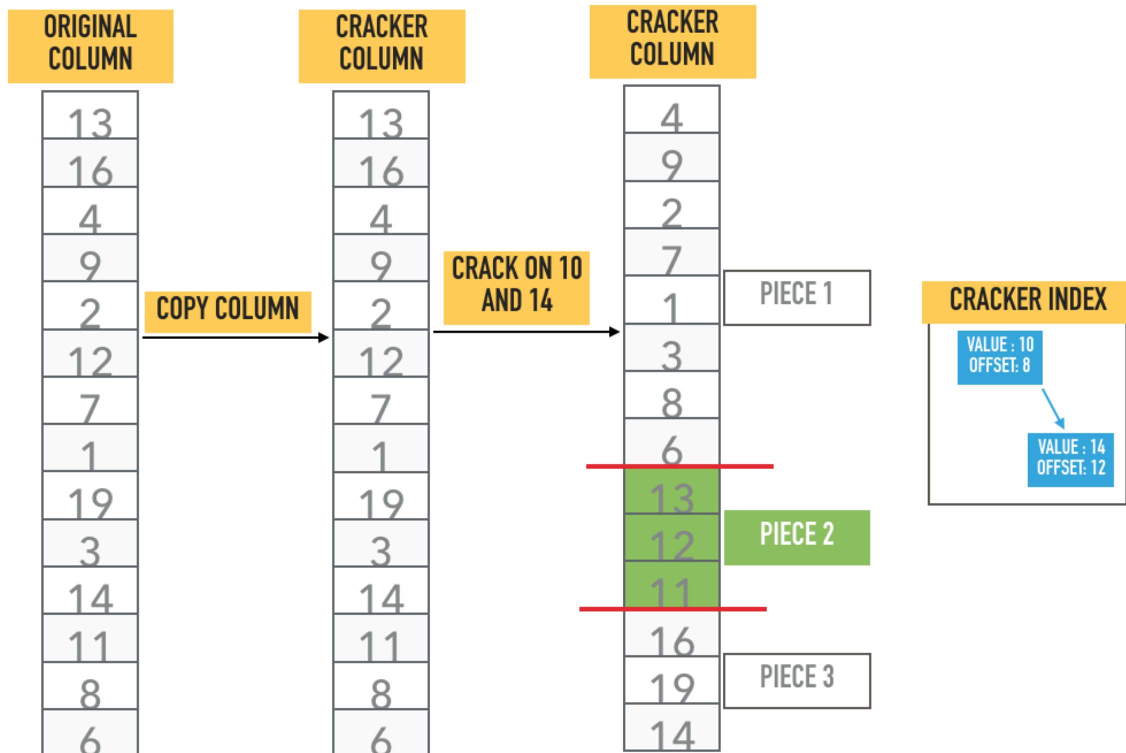


Figure 2-4: Standard Cracking executing filter $C > 10$ and $C < 14$.

Figure 2-4 depicts an example of standard cracking executing a query that requests all values higher than 10 and lower than 14, and the original column has no index yet. When this query is executed, it triggers the first step of database cracking, which performs a full copy of the original column. After copying it to a structure called *cracker column*, it performs two quick-sort iterations using, as quick-sort pivots, the query predicates 10 and 14. This results in a cracker column cracked into three pieces. Where *Piece 1* has all elements up to our first query predicate (i.e., 10), *Piece 2* all elements between our query predicates (i.e., 10 and 14), and *Piece 3* with all elements above or equal to the second predicate (i.e., 14). The information regarding the pieces (i.e., where each piece start and the highest element within that piece is stored in an AVL-Tree [6] (i.e., a binary search tree with self-balancing properties) called *cracker index*. When the next query is executed, the system can already take advantage of this index (e.g., if a query only has one filter $c > 18$, only *Piece 3* needs to be checked). After the first query, the pieces are refined even further by performing new quick-sort iterations with pivots equal to the currently executing filter predicates.

While database cracking accomplishes its mission of constructing an index as a by-product of querying, it suffers from several problems that make it unsuitable for interactive data analysis: (1) it adds a significant overhead over naive scans in the first

iterations of the algorithm, (2) the performance of cracking is not robust, as sudden changes in workload cause spikes in performance, and (3) convergence towards a full index is slow and workload-dependent.

There is a large body of work on extending and improving database cracking. These improvements include better convergence towards a full index [21, 50], more predictable performance [49, 26], more efficient tuple reconstruction [35, 37, 50], better CPU utilization [46], other cracking engines [47, 25], predictive query processing [57], using modern hardware to boost query execution [39], using mediocre elements as cracking pivots [62], creating multidimensional adaptive indexes [45], generalizing database cracking [49] and handling updates [34, 29].

CHAPTER 3

Progressive Indexing

1 Introduction

Data scientists perform exploratory data analysis to discover unexpected patterns in large collections of data. This process is done with a hypothesis-driven trial-and-error approach [52]. They query data segments that could potentially provide insights, test their hypothesis, and either zoom in on the same segment or move to a different one depending on the insights gained.

Fast responses to queries are crucial to allow for interactive data exploration. The study by Liu et al. [41] shows that any delay larger than 500ms (the "interactivity threshold") significantly reduces the rate at which users make observations and generate hypotheses. When dealing with small data sets, providing answers within this interactivity threshold is possible without utilizing indexes. However, exploratory data analysis is often performed on larger data sets as well. In these scenarios, indexes are required to speed up query response times.

Index creation is one of the major difficult decisions in database schema design [15]. Based on the expected workload, the database administrator (DBA) needs to decide whether creating a specific index is worth the overhead in creating and maintaining it. Creating indexes up-front is especially challenging in exploratory and interactive data analysis, where queries are not known in advance, workload patterns change frequently, and interactive responses are required. In these scenarios, data scientists load their data and immediately want to start querying it without waiting for index

construction. In addition, it is also not certain whether or not creating an index is worth the investment at all. We cannot be sure that the column will be queried frequently enough for the large initial investment of creating a full index to pay off.

In spite of these challenges, indexing remains crucial for improving database performance. When no indexes are present, even simple point and range selections require expensive full table scans. When these operations are performed on large data sets, indexes are essential to ensure interactive query response times. Two main strategies aim to release the DBA of having to choose which indexes to create manually.

(1) Automated index selection techniques [1, 14, 58, 23, 13, 11, 44, 53] accomplish this by attempting to find the optimal set of indexes given a query workload, taking into account the benefits of having an index versus the added costs of creating the entire index and maintaining it during modifications to the database. However, these techniques require a priori knowledge of the expected workloads and do not work well when the workload is not known or changes frequently. Hence they are not suitable for interactive data exploration.

(2) Adaptive Indexing techniques such as Database Cracking [36, 21, 50, 49, 26, 35, 37, 47, 46, 25, 34, 29] are a more promising solution. They focus on automatically and incrementally building an index as a side effect of querying the data. An index for a column is only initiated when it is first queried. As the column is queried more, the index is refined until it eventually approaches a full index's performance. In this way, the cost of creating an index is smeared out over the cost of querying the data many times, though not necessarily equally, and there is a smaller initial overhead for starting the index creation. However, since the index is refined only in the areas targeted by the workload, convergence to a full index is not guaranteed, and partitions can have different sizes. The query's performance degrades when a less refined part of the index is queried, resulting in performance spikes whenever the workload changes.

In this chapter, we introduce a new incremental indexing technique called *Progressive Indexing*. It differs from other indexing solutions in that the indexing budget (i.e., the amount of time spent on index creation and refinement) can be controlled. We provide two indexing budget flavors: a fixed indexing budget, where the user defines a fixed amount of time to spend on indexing per query, and an adaptive indexing budget, where the indexing budget is adapted so that the total time spent on query execution remains constant. We refer to the fixed indexing budget as Progressive Indexing and the adaptive indexing budget as Greedy Progressive Indexing. As a result, both Progressive Indexing and Greedy Progressive Indexing complements existing automatic indexing techniques by offering predictable performance and deterministic convergence

independent of the workload.

1.1 Contributions

The main contributions of this chapter are:

- We introduce several novels Progressive Indexing techniques and investigate their performance, convergence, and robustness in the face of various realistic synthetic workload patterns and real-life workloads.
- We provide a cost model for each of the Progressive Indexing techniques. The cost models are used to adapt the indexing budget automatically.
- We experimentally verify that the Progressive Indexing techniques we propose provide robust and predictable performance and convergence regardless of the workload or data distribution.
- We provide a decision tree to assist in choosing an indexing technique for a given scenario.
- We provide Open-Source implementations of each of the techniques we describe and their benchmarks.[1]

1.2 Outline

This chapter is organized as follows. Section 2 depicts related research performed on automatic/adaptive index creation. In Section 3, we describe our novel Progressive Indexing techniques and discuss their benefits and drawbacks. Section 4 describes the cost-models used to adapt our indexing budget automatically. In Section 5, we perform an experimental evaluation of each of the novel methods we introduce, and we compare them against Adaptive Indexing techniques. Finally, in Section 6 we draw our conclusions and present a decision tree to assist in choosing which Progressive Indexing technique to use.

2 Related Work

In this section, we discuss the state-of-the-art of Adaptive Indexing in terms of performance and robustness. Section 2.1 we discuss possible cracking kernels to get

[1] Our implementations and benchmarks are available at `https://github.com/pdet/ProgressiveIndexing`

the partitioning as fast as possible. In Section 2.2 we discuss three different Adaptive Indexing algorithms that attempt to improve cracking's robustness problem.

2.1 Cracking Kernels

A cracking kernel [47, 25] is the central part of how the partitioning of a piece is done. This section focuses on two partitioning kernels. First, we present the branching kernel, which uses if-else clauses to decide when to swap elements. Second, we describe the predicated kernel that uses predication to avoid branch mispredictions.

Branching Kernel

The branching kernel is the one used in the Standard Cracking implementation and has a clear inspiration from quicksort's partitioning [27]. Listing 1 depicts the kernel for the integer data type. It receives as input the array, the pivot, and the boundaries of the partition *posL* and *posR*. The algorithm, inspects all vector elements, and increase *posL* in case the element *data[posL]* is less than the pivot and increases *posR* in case the element *data[posR]* is greater than or equal to the pivot. In other words, it simply moves the cursors if the elements are already in the correct position in reference to the pivot. If it finds both *data[posL]* and *data[posR]* that are not in the correct position, it swaps them and move the cursors. The main problem with this kernel is that swapping the data in the if-else clauses causes an increase in branch mispredictions and an overall decrease in performance, as demonstrated in Boncz et al. [10].

Listing 1 Branching Kernel

```
1  void branching_kernel(int& data, int pivot, size_t posL, size_t posR){
2      while (posL < posR){
3          if (data[posL] < pivot){
4              posL++;
5          }
6          else if (data[posR]>= pivot{
7              posR--;
8          }
9          else{
10             swap(data[posL++],data[posR--])
11         }
12     }
13 }
```

Predicated Kernel

The predicated kernel removes the if-else clauses to avoid branch misprediction costs. Listing 2 demonstrates the predicated kernel for integers. Like the branching kernel, we iterate over the vector. In lines 3 and 4, we store the values we will inspect in this iteration. Lines 5,6, and 7 store integers that inform if a given element must be swapped. For example, if *data[posL]* is lower than *pivot*, that means that *data[posL]* is already in its correct position, hence the *start_has_to_swap* variable will hold 0. Lines 8 - 11 effectively swap the data and modify the cursors with respect on the information in the *start_has_to_swap*, *end_has_to_swap*, and *has_to_swap* variables. The predicated kernel has an extremely predictable cost since it will always execute the same code, independent of branches.

Listing 2 Predicated Kernel

```
void predicated_kernel(int& data, int pivot, size_t posL, size_t posR){
    while (posL < posR){
        int l_value = data[posL];
        int r_value = data[posR];
        int start_has_to_swap = l_value >= pivot;
        int end_has_to_swap = r_value < pivot;
        int has_to_swap = start_has_to_swap * end_has_to_swap;
        data[posR] = !has_to_swap * l_value + has_to_swap * r_value;
        data[posL] = !has_to_swap * r_value + has_to_swap * l_value;
        posL+= !start_has_to_swap + has_to_swap;
        posR -= !end_has_to_swap + has_to_swap;
    }
}
```

2.2 Adaptive Indexing for Robustness

Stochastic Cracking [26]

Stochastic Cracking minimizes the unforeseen performance issues from cracking. Instead of using query predicates as pivots, a random element from the to-be-cracked piece is used as the partitioning pivot. Hence this decreases the workload dependency from cracking.

Figure 3-1 depicts an example of Stochastic Cracking. From our example, the cracker column is initially unpartitioned. When executing the first query that requests all elements greater than 15, a random element from the column is selected as a pivot.

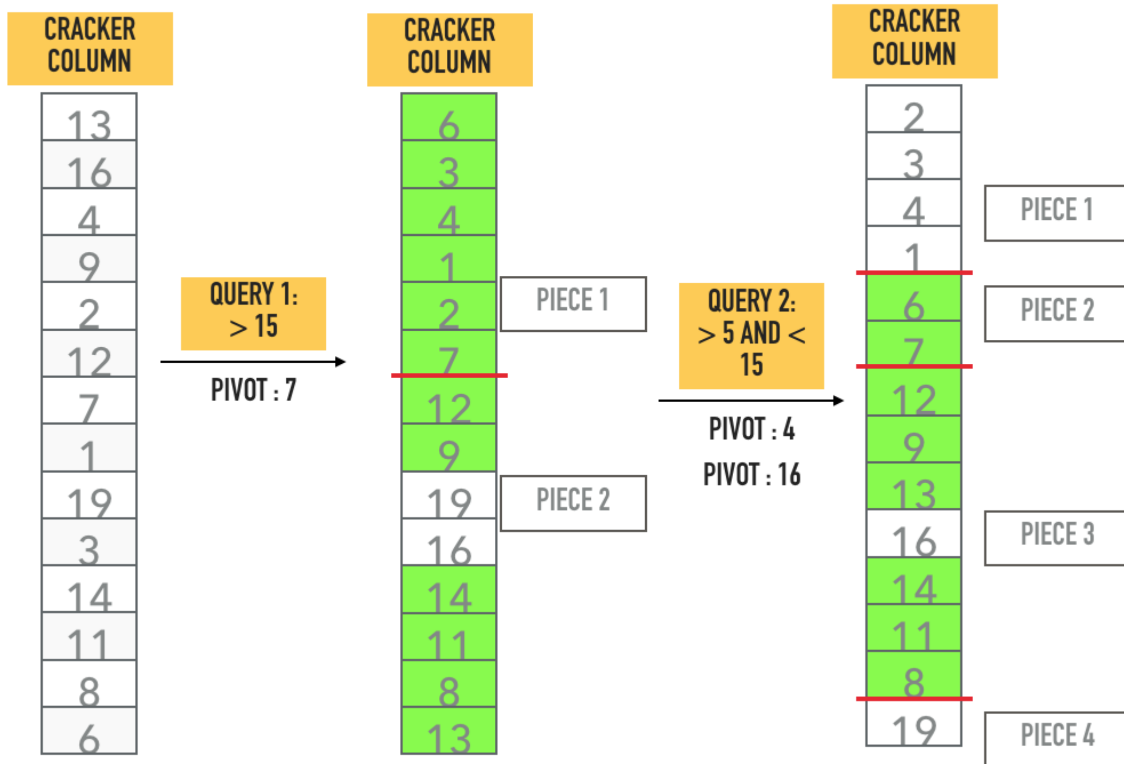


Figure 3-1: Standard Cracking executing two queries.

In our example, the element 7, the column is then partitioned around 7, and both pieces must be scanned to answer the query. When query 2 is executed requesting all elements between 5 and 15, *Piece 1* is pivoted with an element within the piece, in this case, 4, and the same happens with *Piece 2*, with pivot 16 being selected to partition it. After finishing the partition, only piece 2 (i.e., all elements over 4) and piece 3 (i.e., all elements higher than 7 and lower or equal to 16) must be scanned.

Not using the filter predicates as query pivots can result in the execution engine reading more data than necessary even after the partitioning for that query. However, sudden changes in the workload pattern will not have the same impact as in Standard Cracking.

Progressive Stochastic Cracking [26]

Progressive Stochastic Cracking progressively performs Stochastic Cracking. It takes two input parameters, the size of the L2 cache and the number of swaps allowed in one iteration (i.e., a percentage of the total column size). When performing Stochastic Cracking, Progressive Stochastic Cracking will only perform at most the maximum allowed number of swaps on pieces larger than the L2 cache. If the piece fits into the

L2 cache, it will always perform a complete crack of the piece.

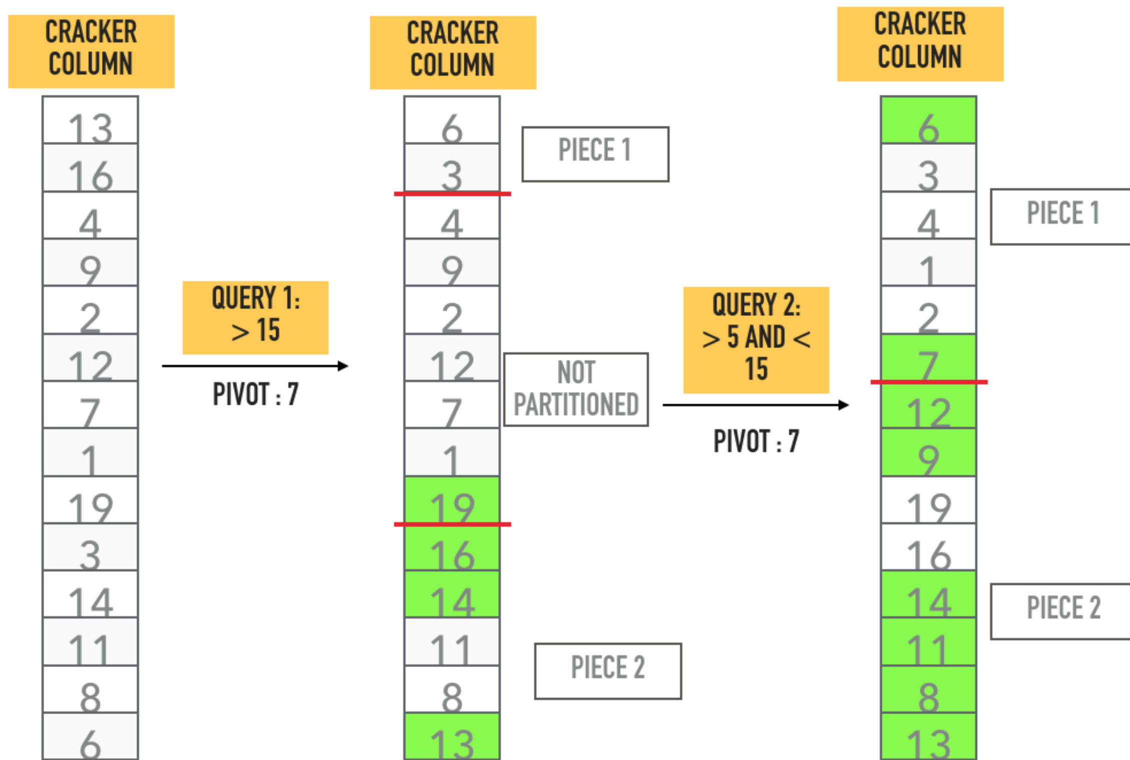


Figure 3-2: Progressive Stochastic Cracking with maximum swaps = 2 and L2 Cache Size = 8kb.

Figure 3-2 depicts an example of Progressive Stochastic Cracking, where the L2 Cache Size fits two integers and the at most two swaps can be performed per query. Like Stochastic Cracking, the pivots are also selected randomly from within the piece that will be partitioned. In our first query, the pivot chosen is 7. The difference is that when executing this query, we stop pivoting after swapping two elements. When executing *Query 2*, we finish the partition with pivot 7 before picking new pivots.

Coarse-Granular Index [50]

The Coarse-Granular Index improves Stochastic Cracking's robustness by creating k partitions when the first query is executed using equal-width binning. It also allows for creating any number of partitions instead of limiting the number of partitions to two, letting the DBA decide on k, choosing between the trade-off of the higher cost of the first query versus building a more robust index.

Figure 3-3 depicts an example of the Coarse-Granular Index set to create four partitions. When executing the first query, the algorithm will perform 3 cracking

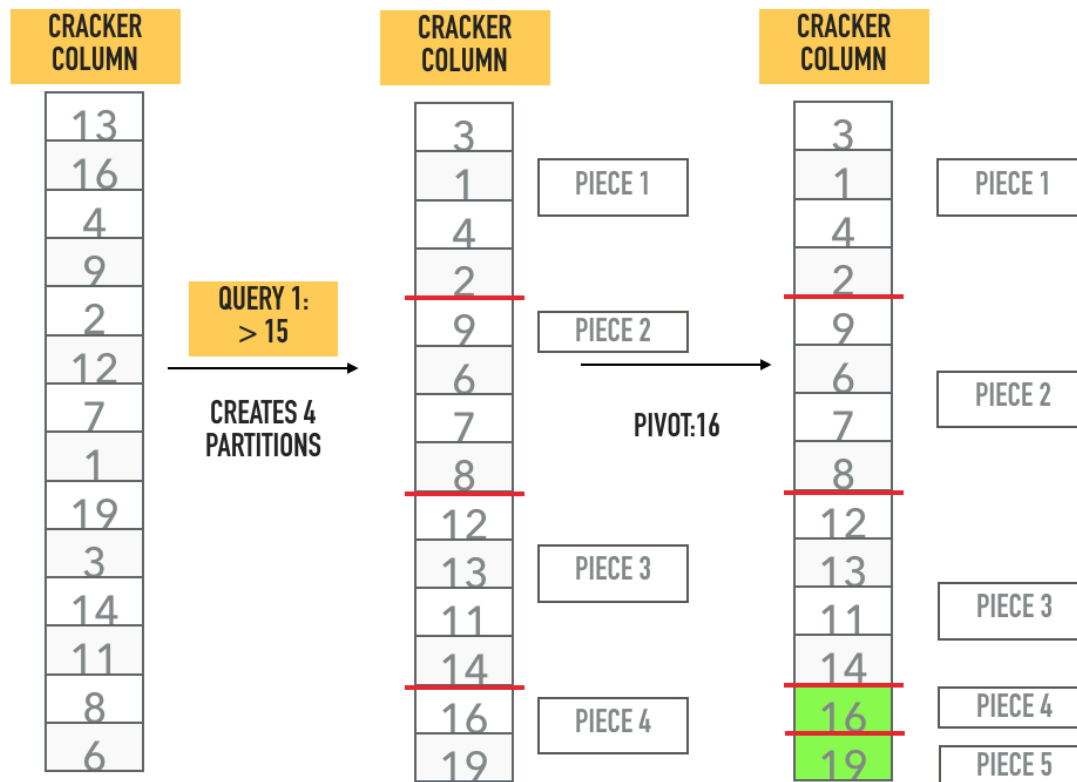


Figure 3-3: Coarse Granular-Index creating $k = 4$ partitions in the first query.

iterations from the equi-width binning (i.e., since our data goes from 1 to 20, that means the pivots will be 5,10, and 15). After it, a standard Stochastic Cracking iteration happens. At that point, it is only necessary to check *Piece 4* since it holds all elements over 15. A random pivot from within the piece is selected, in this case, 16, and the query answer is produced.

Adaptive Adaptive Indexing [49]

Adaptive Adaptive Indexing is a general-purpose algorithm for Adaptive Indexing. It has multiple parameters tuned to mimic the data access of different Adaptive Indexing techniques (e.g., Database Cracking, Sideways Cracking, Hybrid Cracking). It also uses radix partitioning and exploits software-managed buffers using nontemporal streaming stores to achieve better performance [51].

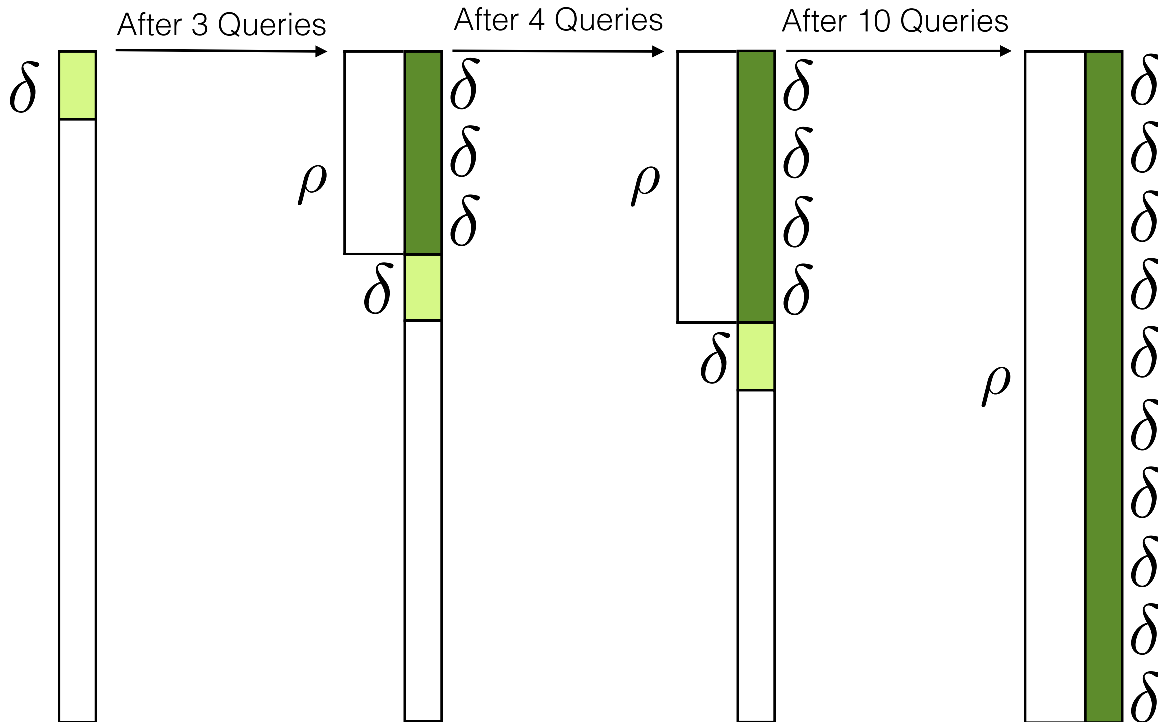


Figure 3-4: Creation phase of Progressive Indexing.

3 Progressive Indexing

In this section, we introduce *Progressive Indexing.* The core features of Progressive Indexing are that (1) the indexing overhead per query is controllable, both in terms of time and memory requirements, (2) it offers robust performance and deterministic convergence regardless of the underlying data distribution, workload patterns, or query selectivity, and (3) the indexing budget can be automatically tuned so more expensive queries spend less extra time on indexing while cheaper queries spend more. To allow for robust query execution times regardless of the data, we avoid branches in the code and use predication when possible [48, 10].

As a result of the small initial cost, Progressive Indexing occurs without significantly impacting worst-case query performance. Even if the column is only queried once, only a small penalty is incurred. On the other hand, if the column is queried hundreds of times, the index will reliably converge towards a full index, and queries will be answered at the same speed as with an a-priori built full index.

All Progressive Indexing algorithms progress through three canonical phases to eventually converge to a full B+-tree index: the *creation phase*, the *refinement phase*, and the *consolidation phase.* Each phase's work can be divided between multiple queries, keeping the extra indexing effort per query strictly limited.

Creation Phase. The creation phase progressively builds an initial "crude" version of the index by adding another δ fraction of the original column to the index with each query. Query execution during the creation phase is performed in three steps(visualized in Figure 3-4):

1. Perform an index lookup on the ρ fraction of the data that has already been indexed;
2. Scan the not-yet-indexed $1 - \rho$ fraction of the original column;
   and while doing so,
3. Expand the index by another δ fraction of the total column.

As the index grows and the fraction ρ of the indexed data increases, an ever-smaller fraction of the base column has to be scanned, progressively improving query performance. Once all the base column data has been added to the index, the creation phase is followed by the refinement phase.

Refinement Phase. With the base column no longer required to answer queries, we only perform lookups into the index to answer queries. While doing these lookups, we further refine the index, progressively converging towards a fully ordered index. In the refinement phase, we focus on refining parts of the index required for query processing. After these parts have been refined, the refinement process starts processing the neighboring parts. Once the index is fully ordered, the refinement phase is followed by the consolidation phase.

Consolidation Phase. With the index fully ordered, we progressively construct a B+-tree from it since a B+-Tree provides better data locality and thus is more efficient than binary search when executing very selective queries. Once the B+-tree is completed, we use it exclusively to answer all subsequent queries. The consolidation phase is the same for all progressive algorithms. All algorithms end their refinement phase with a sorted array. The B+-tree is then constructed on top of that sorted array in a bottom-up fashion. Figure 3-5 depicts an example of the construction phase for Progressive Quicksort in the right-most part of the figure labeled *Consolidation*. In this example, the B+-Tree stored 4 elements per node. Hence we start constructing the last level of the inner nodes pointing to one element every four elements. In this case, the B+-Tree nicely ends with one inner node that is also the root. However, if there were more elements, we would fully construct this level, link all nodes, and proceed to the upper level and repeat this strategy.

In the following section, we discuss the details of four different Progressive Indexing implementations. Section 3.1 describes Progressive Quicksort as a progressive version of quicksort, aiming to achieve good performance independent of query patterns and data distributions. In Section 3.2 we present Progressive Radixsort - Most Significant Digit as the radixsort algorithm this index is based on, we expect good performance over uniform distributions. In Section 3.3 we present Progressive Bucketsort, inspired by bucketsort equi-height, which is expected to present excellent performance with highly skewed data distributions. Finally, in Section 3.4 we present Progressive Radixsort - Least Significant Digit, where we aim to optimize for workloads that contain only point queries.

3.1 Progressive Quicksort

Figure 3-5 depicts snapshots of the creation phase, the refinement phase, and the consolidation phase of Progressive Quicksort. We discuss the creation and refinement phases in detail in the following paragraphs.

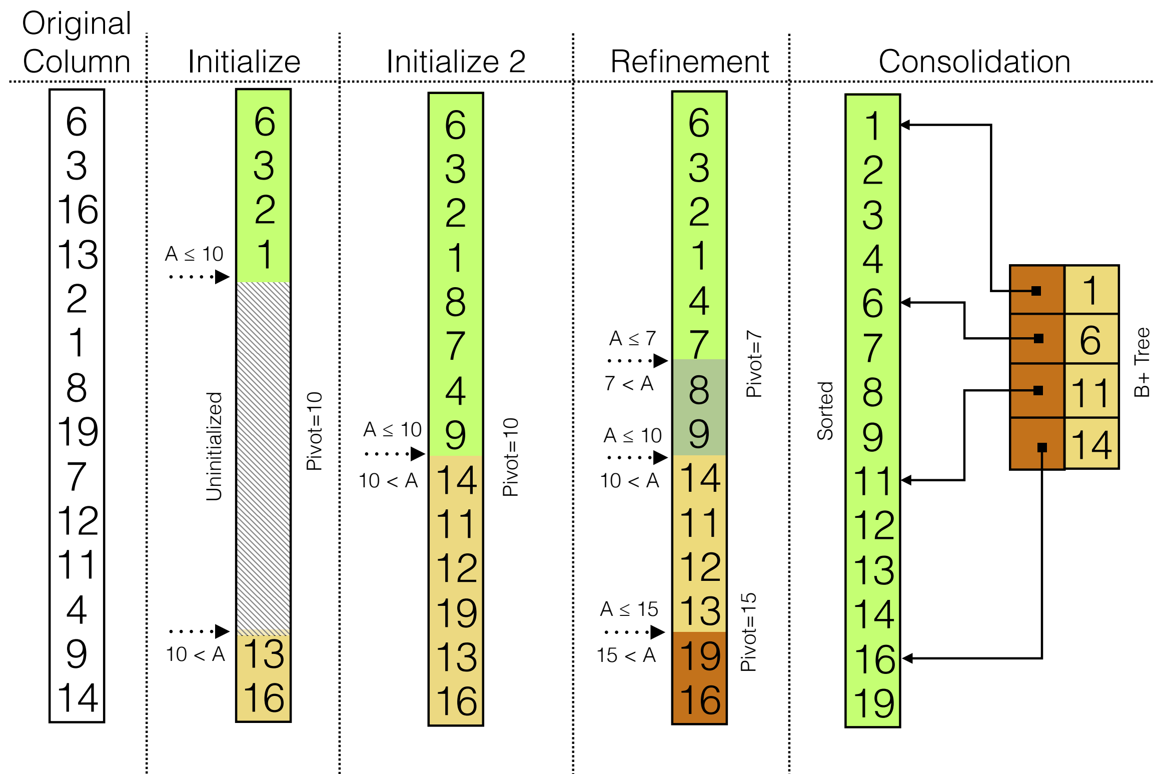


Figure 3-5: Progressive Quicksort.

Creation Phase

In the first iteration, we allocate an uninitialized column of the same size as the original column and select a pivot. The pivot is selected by taking the average value of the smallest and largest value of the column. In Figure 3-5, pivot 10 is the average of 1 and 19. If sufficient statistics are available, the median value of the column could be used instead. Unlike Adaptive Indexing, the pivot selection is not impacted by the query predicates. We then scan the original column and copy the first $N * \delta$ elements to either the top or bottom of the index, depending on their relation to the pivot. In this step, we also search for any elements that fulfill the query predicate and afterward scan the not-yet-indexed $1 - \rho$ fraction of the column to compute the complete answer to the query. In subsequent iterations, we scan either the top, bottom, or both parts of the index based on how the query predicate relates to the chosen pivot.

Refinement Phase

We refine the index by recursively continuing the quicksort in-place in the separate sections. The refinement consists of swapping elements in-place inside the index around the pivots of the different segments. When the pivoting of a segment is completed, we recursively continue the quicksort in the child segments. We maintain a binary tree of the pivot points. In this tree's nodes, we keep track of the pivot points and how far along the pivoting process we are. To do an index lookup, we use this binary tree to find the array sections that could match the query predicate and only scan those, effectively reducing the amount of data to be accessed even when the full pivoting has not been completed yet.

When we reach a node that is smaller than the L1 cache, we sort the entire node instead of recursing any further. After sorting a node entirely, we mark it as sorted. When two children of a node are sorted, the entire node itself is sorted, and we can prune the child nodes. As the algorithm progresses, leaf nodes will keep on being sorted and pruned until only a single fully sorted array remains.

3.2 Progressive Radixsort (MSD)

Figure 3-6 depicts snapshots of the creation phase, the refinement phase, of Progressive Radixsort (MSD). We discuss both phases in detail in the following paragraphs.

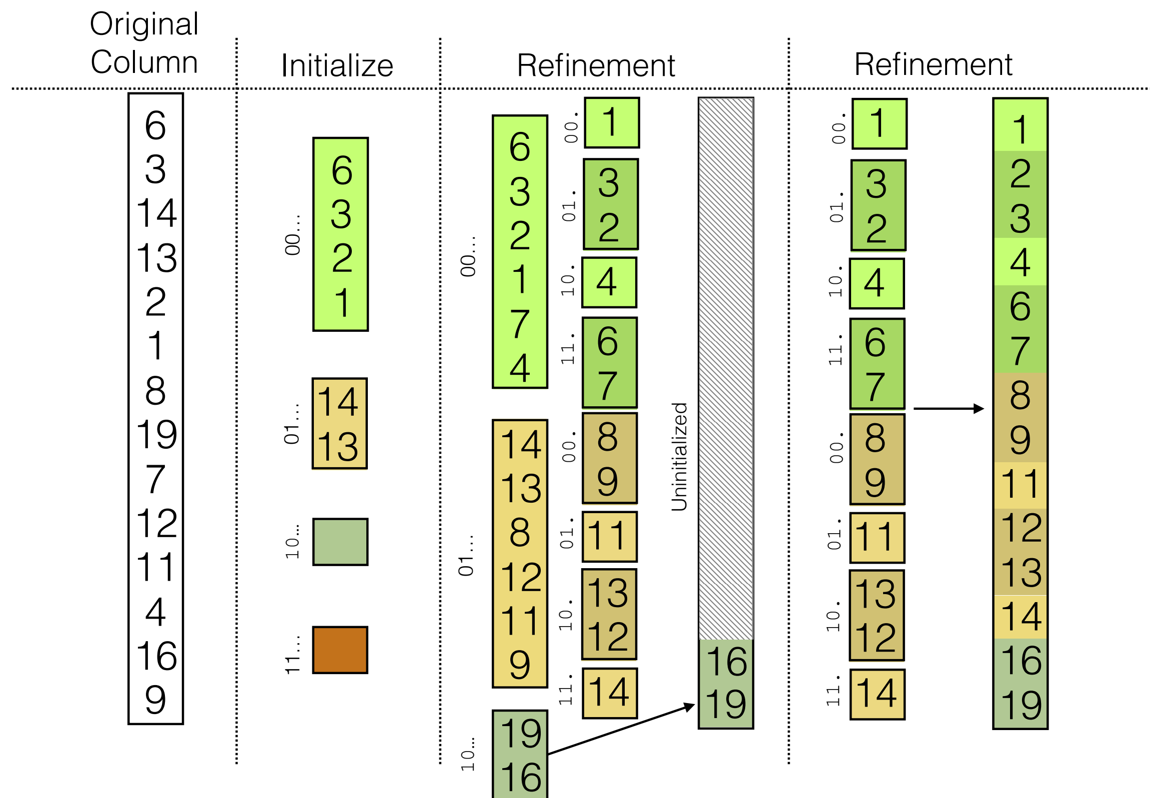


Figure 3-6: Progressive Radixsort (MSD).

Creation Phase

In the creation phase of Progressive Radixsort, we perform the radixsort partitioning into buckets located in separate memory regions. We start by allocating b empty buckets. Then, while scanning the original column, we place $N * \delta$ elements into the buckets based on their most significant $\log_2 b$ bits. We then scan the remaining $1 - \rho$ fraction of the base column. In subsequent iterations, we scan the $[0, b]$ buckets that could potentially contain elements matching the query predicate to answer the query in addition to scanning the remainder of the base column.

Bucket Count. Radix clustering performs a random memory access pattern that randomly writes in b output buckets. To avoid excessive cache- and TLB-misses, assuming that each bucket is at least of the size of a memory page, the number b of buckets, and thus the number of randomly accessed memory pages, should not exceed the number of cache lines and TLB entries, whichever is smaller [9]. Since our machine has 512 L1 cache lines and 64 TLB entries, we use $b = 64$ buckets.

Bucket Layout. To avoid allocating large regions of sequential data for every bucket, the buckets are implemented as a linked list of blocks of memory that each

hold up to s_b elements. When a block is filled, another block is added to the list, and elements will be written to that block. This adds some overhead over sequential reads/writes as for every s_b elements there will be a memory allocation and random access, and for every element that is added, the bounds of the current block have to be checked.

Refinement Phase

In the refinement phase, all elements in the original column have been appended to the buckets. In this phase, we recursively partition by the next set of $\log_2 b$ most significant digits. For each of the buckets, this results in creating another set of b buckets in each of the refinement phases, for a total of $b * b$ buckets in the second phase. To avoid the overhead of managing these buckets to become bigger than the overhead of actually performing the radix partitioning, we avoid re-partitioning buckets that fit into the L1 cache and instead immediately insert the values of these buckets in sorted order into the final sorted array, as shown in Figure 3-6. As the buckets themselves are ordered (i.e., for two buckets b_i and b_{i+1}, we know $e_i < e_{i+1} \forall e_i \in b_i, e_{i+1} \in b_{i+1}$), we know the position of each bucket in the final sorted array without having to consider any elements in the other buckets.

We keep track of the buckets using a tree in which the nodes point towards either the leaf buckets or towards a position in the final sorted array if the leaf buckets have already been merged in there. This tree is used to answer queries on the intermediate structure. When we get a query, we look up which buckets we have to scan based on the query predicates' most significant bits. We then scan the buckets or the final index, where required.

When the first iteration of the refinement phase is completed, we recursively continue with the next set of $\log_2 b$ most significant digits until all the elements have been merged and sorted into the final index. At that point, we construct our B+-tree index from the single fully sorted array.

3.3 Progressive Bucktersort

Progressive Bucketsort (Equi-Height) is very similar to Progressive Radixsort (MSD). The main difference is in the way the initial partitions (buckets) are determined. Instead of radix clustering, which is fast but yields equally sized partitions only with uniform data distributions, we perform a value-based range partitioning to yield equally sized partitions also with skewed data, at the expense that determining the

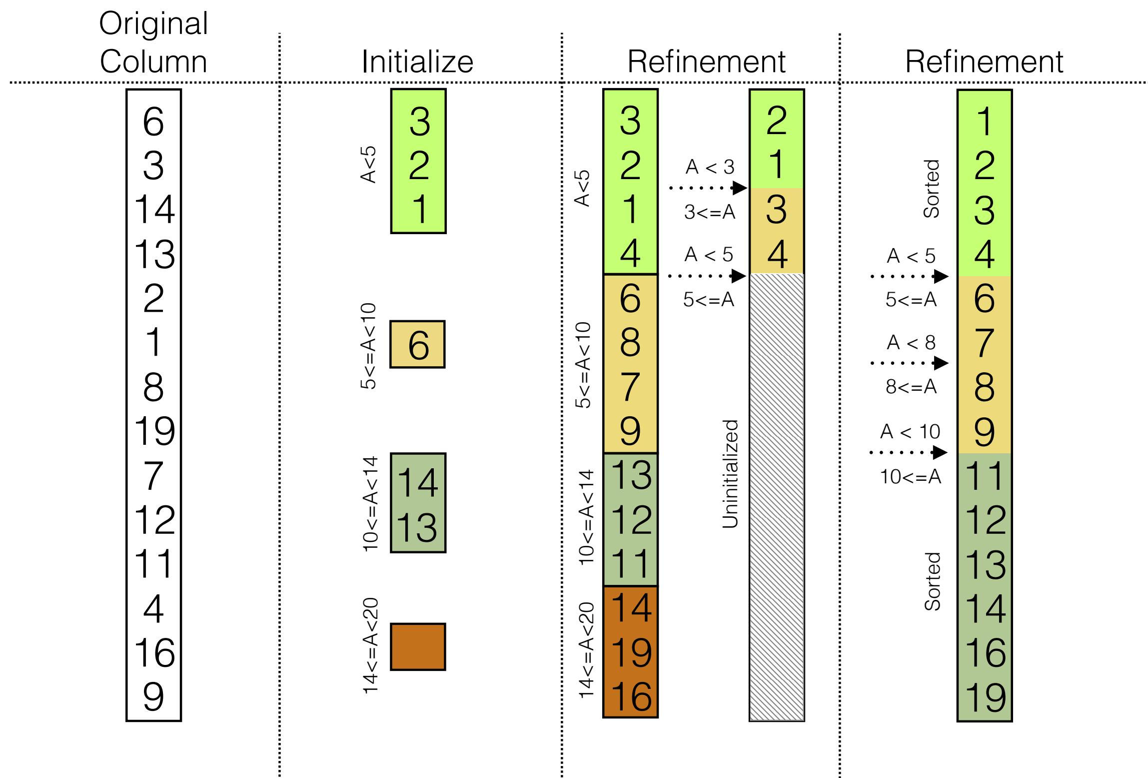


Figure 3-7: Progressive Bucket Sort

bucket that a value belongs to is more expensive. Figure 3-7 depicts a snapshot of the creation phase and two snapshots of the refinement phase. In the following, we discuss these two phases in detail.

Bucket Count. To optimize for writing and reading from the buckets, our implementation of Progressive Bucketsort uses 64 buckets, as discussed in Section 3.2.

Creation Phase

Progressive Bucketsort operates in a very similar way to Progressive Radixsort (MSD). Instead of choosing the bucket an element belongs to based only on the most significant bits, the bucket is chosen based on a set of bounds that more-or-less evenly divide the set elements into the separate buckets. These bounds can be obtained either in the scan to answer the first query or from existing statistics in the database (e.g., a histogram).

Refinement Phase

In the refinement phase, all elements in the original column have been appended to the buckets. We then merge the buckets into a single sorted array. Unlike with Progressive Radixsort (MSD), we do not recursively keep on using Progressive Bucketsort. This is because the overhead of finding and maintaining the equi-height bounds for each sub-bucket is too large. Instead, we sort the individual buckets into the final sorted list using Progressive Quicksort. Using a progressive algorithm to sort individual buckets protects us from performance spikes caused by sorting large buckets.

The buckets are merged into the final sorted index in order. As such, we always have a single iteration of Progressive Quicksort active at a time in which we are performing swaps. After all the buckets have been merged and sorted into the final index, we have a single fully sorted array from which we can construct our B+-tree index.

3.4 Progressive Radixsort (LSD)

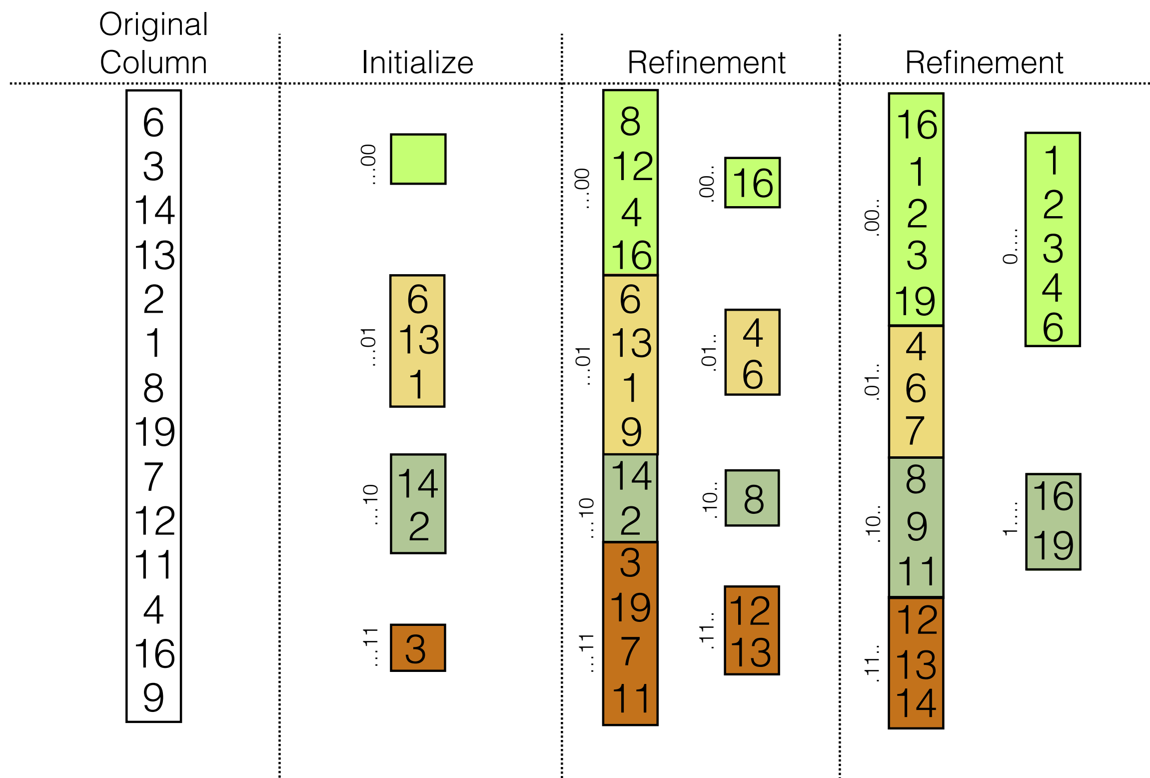


Figure 3-8: Progressive Radixsort (LSD).

Progressive Radixsort Least Significant Digits (LSD) performs a progressive radix clustering on the least significant bits during the creation and refinement phase. Figure 3-8 depicts a snapshot of the creation phase and two snapshots of the refinement phase. In the following, we discuss these two phases in detail.

Bucket Count. To optimize for writing and reading from the buckets, our implementation of Progressive Radixsort (LSD) uses 64 buckets, as discussed in Section 3.2.

Creation Phase

This algorithm's creation phase is similar to the creation phase of Progressive Radixsort (MSD), except that we partition elements based on the least-significant bits instead of the most-significant bits. We can use the buckets created to speed up point queries because we only need to scan the bucket in which the query value falls. However, unlike the buckets created for the Progressive Radixsort (MSD) and Progressive Bucketsort, these intermediate buckets cannot be used to speed up range queries in many situations. Because the elements are inserted based on their least-significant bits, the buckets do not form a value-based range-partitioning of the data. Consequently, we will have to scan many buckets, depending on the domain covered by the range query.

Refinement Phase

In the refinement phase, we move elements from the current set of buckets to a new set of buckets based on the next set of significant bits. We repeat this process until the column is sorted. How many iterations this takes depends on the bucket count and the column's value domain, which we obtain from the $[min, max]$ values. We can compute the amount of required iterations with the formula $\lceil log_2(max - min)/log_2(b) \rceil$. For example, for a column with values in the range of $[0, 2^{16})$ and 64 buckets, the amount of iterations required before convergence is $\lceil log_2(2^{16})/log_2(64) \rceil = 3$.

4 Greedy Progressive Indexing

The value of δ determines how much time is spent constructing the index and hence determines the indexing budget. Greedy Progressive Indexing allows the user to select between setting either a fixed indexing budget or an adaptive indexing budget. For the fixed indexing budget, the user provides the desired indexing budget t_{budget} to

Table 3.1: Parameters for Greedy Progressive Quicksort Cost Model.

System	ω	cost of sequential page read (s)
	κ	cost of sequential page write (s)
	ϕ	cost of random page access (s)
	γ	elements per page
Data set	N	number of elements in the data set
& Query	α	% of data scanned in partial index
Index	δ	% of data to-be-indexed
	ρ	% of data already indexed
Progressive Quicksort	h	height of the binary search tree
Progressive	b	number of buckets
Radixsort	s_b	max elements per bucket block
	τ	cost of memory allocation (s)
B+-Tree	β	tree fanout

spend on indexing for the first query. We then select the value of δ based on this budget and use that δ for the remainder of the workload. The adaptive indexing budget allows the user to specify the desired indexing budget for the first query t_{budget}. The first query will then execute in time $t_{adaptive} = t_{scan} + t_{budget}$. After the first query, the value of δ will be adapted such that the query cost will stay equivalent to $t_{adaptive}$ until the index is converged.

Cost Model. We use a cost model to determine how much time we can spend on indexing when working with the adaptive indexing budget. The cost model takes into account the query predicates, the selectivity of the query and the state of the index in a way that is not sensitive to different data distributions or querying patterns and does not rely on having any statistics about the data available.

4.1 Greedy Progressive Quicksort

The parameters of the Greedy Progressive Quicksort cost model are summarized in Table 3.1.

Creation Phase

The total time taken in the creation phase is the sum of (1) the scan time of the base table, (2) the index lookup time, and (3) the additional indexing time. The scan time is given by multiplying the number of pages we need to scan ($\frac{N}{\gamma}$) by the amount of time it takes for a sequential page access (ω), resulting in $t_{scan} = \omega * \frac{N}{\gamma}$. The pivoting time (i.e., index construction time) consists of scanning the base table pages and writing the pivoted elements to the result array. The pivoting time is therefore

obtained by multiplying the time it takes to scan and write a page sequentially $(\kappa+\omega)$ by the number of pages we need to write, resulting in $t_{pivot} = (\kappa + \omega) * \frac{N}{\gamma}$.

The total time taken for the initial indexing process is given by multiplying the scan time by the fraction of the base table we need to scan. Initially, we need to scan the entire base table, but as the fraction of indexed data (ρ) increases, we need to scan less. Instead, we scan the index to answer the query. The amount of data we need to scan in the index depends on how the query predicates relate to the pivot. The fraction of data that we need to scan is given by α and can be computed for a given set of query predicates. The total fraction of the data that we scan is $1 - \rho + \alpha - \delta$. The fraction of the data that we index in each step is δ. Hence the total time taken is given by $t_{total} = (1 - \rho + \alpha - \delta) * t_{scan} + \delta * t_{pivot}$.

Indexing Budget. In this phase, we set delta such that $\delta = \frac{t_{budget}}{t_{pivot}}$. For the fixed indexing budget, we select this δ for the first query and keep on using this δ for the remainder of the workload. For the adaptive indexing budget, we use this formula to select the δ for each query.

Refinement Phase

In the refinement phase, we no longer need to scan the base table. Instead, we only need to scan the fraction α of the data in the index. However, we now need to (1) traverse the binary tree to figure out the bounds of α, and (2) swap elements in-place inside the index instead of sequentially writing them to refine the index. The cost for traversing the binary tree is given by the height of the binary tree h times the cost of a random page access ϕ, resulting in $t_{lookup} = h * \phi$. For the swapping of elements, we perform predicated swapping to allow for a constant cost regardless of how many elements we need to swap. Therefore the cost for swapping is equivalent to the cost of sequential writing (i.e., $t_{swap} = \kappa * \frac{N}{\gamma}$). The total cost in this phase is therefore equivalent to $t_{total} = t_{lookup} + \alpha * t_{scan} + \delta * t_{swap}$.

Indexing Budget. In this phase, we set delta such that $\delta = \frac{t_{budget}}{t_{swap}}$ for the adaptive indexing budget.

Consolidation Phase

In the consolidation phase, we use binary search in the sorted array until the B+-Tree levels are complete. This results in $t_{lookup} = \log_2(n) * \phi$. To construct the B+-Tree, we copy every β element from one level to the next. Therefore the cost of copying the elements is the cost of access a random element from the current level and sequentially

write it to the next, defined by $t_{copy} = N_{copy} * \kappa * \gamma$ The total cost in this phase is equivalent to $t_{total} = t_{lookup} + \alpha * t_{scan} + \delta * t_{copy}$.

Indexing Budget. In this phase, we set delta such that $\delta = \frac{t_{budget}}{t_{copy}}$ for the adaptive indexing budget.

4.2 Greedy Progressive Radixsort (MSD)

This section describes the cost model for both the creation and refinement phases of Greedy Progressive Radixsort (MSD). The consolidation phase follows the same cost model as described in Section 4.1. The parameters are summarized in Table 3.1.

Creation Phase

In the creation phase, the total time taken is the sum of (1) the scan time of the base table, (2) the index lookup time, and (3) the time it takes to add elements to buckets. The scan time of the base table is equivalent to the scan time (t_{scan}) given in Section 3.1. Scanning the buckets for the already indexed data has equivalent performance to performing a sequential scan plus the random accesses we need to perform every s_b elements, hence the scan time of the buckets is equivalent to $t_{bscan} = t_{scan} + \phi * \frac{N}{s_b}$. As we determine which bucket an element belongs to only based on the most significant bits, finding the relevant bucket for an element can be done using a single bitshift. As we chose the bucket count such that all bucket regions can fit in cache, the cost of writing elements to buckets is equivalent to sequentially writing them (κ). We need to perform a memory allocation every s_b entries, which has a cost of τ. This results in a total cost of bucketing equal to $t_{bucket} = (\kappa + \omega) * \frac{N}{\gamma} + \tau * \frac{N}{s_b}$. The total cost is therefore $t_{total} = (1 - \rho - \delta) * t_{scan} + \alpha * t_{bscan} + \delta * t_{bucket}$.

Indexing Budget. In this phase, we set delta such that $\delta = \frac{t_{budget}}{t_{bucket}}$. For the fixed indexing budget, we select this δ for the first query and keep on using this δ for the remainder of the workload. For the adaptive indexing budget, we use this formula to select the δ for each query.

Refinement Phase

The total time taken for a query is the sum of (1) the time taken to scan the required buckets to answer the query predicates and (2) the time taken to perform the radix partitioning of the elements. The time taken to scan the buckets is the same as in the creation phase, $\alpha * t_{bscan}$. The time taken for the radix partitioning is $t_{bucket} = (\kappa + \omega) * \frac{N}{\gamma} + \tau * \frac{N}{s_b}$. The total cost is therefore $t_{total} = \alpha * t_{bscan} + \delta * t_{bucket}$.

Indexing Budget. In this phase, we set delta such that $\delta = \frac{t_{budget}}{t_{bucket}}$ for the adaptive indexing budget.

4.3 Greedy Progressive Bucketsort

In this section, we describe the cost model for the creation phase of Greedy Progressive Bucketsort. The refinement and consolidation phases follow the same cost model described in Section 4.1. The parameters are summarized in Table 3.1.

Creation Phase

In the creation phase, the cost of the algorithm is identical to that of Progressive Radixsort (MSD) except that determining which element a bucket belongs to now requires us to perform a binary search on the bucket boundaries, costing an additional $\log_2 b$ time per element we bucket. This results in the following cost for the initial indexing process $t_{total} = (1 - \rho - \delta) * t_{scan} + \alpha * t_{bscan} + \delta * \log_2 b * t_{bucket}$.

Indexing Budget. In this phase, we set delta such that $\delta = \frac{t_{budget}}{\log_2 b * t_{bucket}}$. For the fixed indexing budget, we select this δ for the first query and keep on using this δ for the remainder of the workload. For the adaptive indexing budget, we use this formula to select the δ for each query.

4.4 Greedy Progressive Radixsort(LSD)

This section describes the cost model for both the creation and refinement phases of Greedy Progressive Radixsort (LSD). The consolidation phase follows the same cost model as described in Section 4.1. The parameters are summarized in Table 3.1.

Creation Phase

The cost model for the Progressive Radixsort (LSD) is also equivalent to the cost model of the Progressive Radixsort (MSD), except the value of α is likely to be higher for range queries (depending on the query predicates) as the elements that answer the query predicate are spread in more buckets. As scanning the buckets is slower than scanning the original column, we also have a fallback when $\alpha == \rho$ we scan the original column instead of using the buckets to answer the query.

Refinement Phase

In this phase, we scan α fraction of the original buckets to answer the query and move δ fraction of the elements into the new set of buckets. This results in the following cost for the refinement process: $t_{total} = \alpha * t_{bscan} + \delta * t_{bucket}$.

Indexing Budget. In this phase, we set delta as $\delta = \frac{t_{budget}}{t_{bucket}}$ for the adaptive indexing budget.

5 Experimental Analysis

In this section, we evaluate the proposed Progressive Indexing methods and the performance characteristics they exhibit. In addition, we provide a comparison of the performance of the proposed methods with Adaptive Indexing methods.

5.1 Setup.

We implemented all our Progressive Indexing algorithms in a stand-alone program written in C++. We included implementations of the Adaptive Indexing algorithms provided by the authors and implemented an adaptive cracking kernel algorithm that picks the most efficient kernel when executing a query, following the decision tree from Haffner et al. [25]. Both the Progressive Indexing algorithms and the existing techniques were compiled with GNU `g++` version 7.2.1 using optimization level `-O3`. All experiments were conducted on a machine equipped with 256 GB of main memory and an 8-core Intel Xeon E5-2650 v2 CPU @ 2.6 GHz with 20480 KB L3 cache.

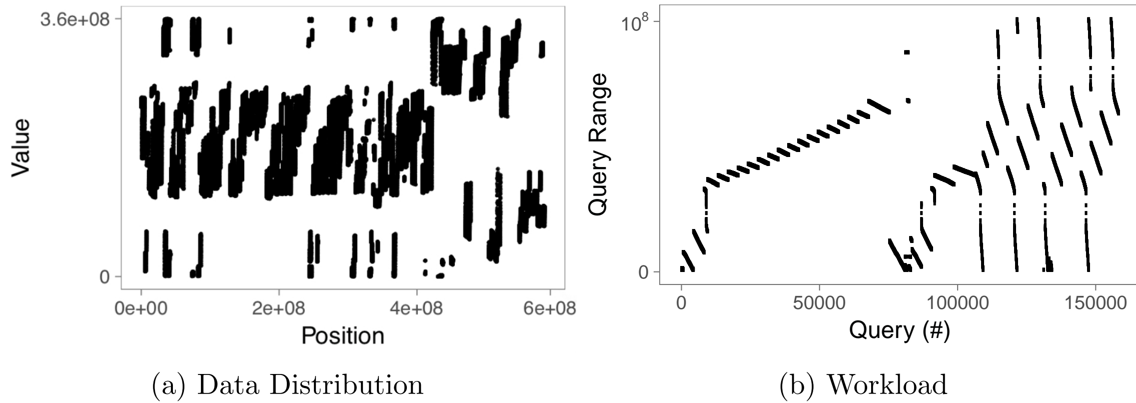
Workloads

In the performance evaluation, we use two data sets a real data set called Skyserver and a synthetic data set.

Skyserver

The Sloan Digital Sky Survey[2] is a project that maps the universe. The data set and interactive data exploration query logs are publicly available via the SkyServer[3] website. Similar to Halim et al. [26] we focus the benchmark on the range queries that are applied on the *Right Ascension* column of the *PhotoObjAll* table. The data set

[2] `https://www.sdss.org/`
[3] `http://skyserver.sdss.org/`



(a) Data Distribution (b) Workload

Figure 3-9: Skyserver

contains almost 600 million tuples, with around 160,000 range queries that focus on specific sections of the domain before moving to different areas. The data and the workload distributions are shown in Figure 3-9.

Synthetic

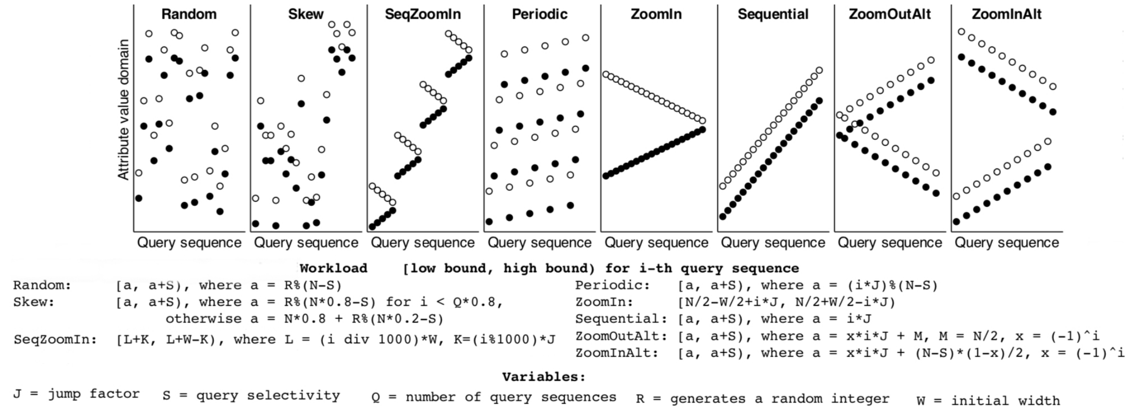


Figure 3-10: Synthetic Workloads [26].

Synthetic. The synthetic data set is composed of two data distributions, consisting of 10^8 or 10^9 8-byte integers distributed in the range of $[0, n)$, i.e., for 10^9 the values are in the range of $[0, 10^9)$. We use two different data sets. The first one is composed of unique integers that are uniformly distributed. In contrast, the second one follows a skewed distribution with non-unique integers where 90% of the data is concentrated in the middle of the $[0, n)$ range. The synthetic workload consists of 10^6 queries in the form `SELECT SUM(R.A) FROM R WHERE R.A BETWEEN` V_1 `AND` V_2. The values for V_1 and V_2 are chosen based on the workload pattern. The different workload patterns and their mathematical description are depicted in Figure 3-10.

5.2 Delta Impact

The δ parameter determines the performance characteristics shown by the Progressive Indexing algorithms. For $\delta = 0$, no indexing is performed, meaning that algorithms resort to performing full scans on the data, never converging to a full index. For $\delta = 1$, the entire creation phase will be completed immediately during the first query execution. Between these two extremes, we are interested in seeing how different values of the δ parameter influence the performance characteristics of the different algorithms.

To measure the impact of different δ parameters on the different algorithms, we execute the SkyServer workload using a $\delta \in [0.005, 1]$. We measure the time taken for the first query, the number of queries until pay-off, the number of queries necessary for full convergence, and the total time spent executing the entire workload.

First Query.

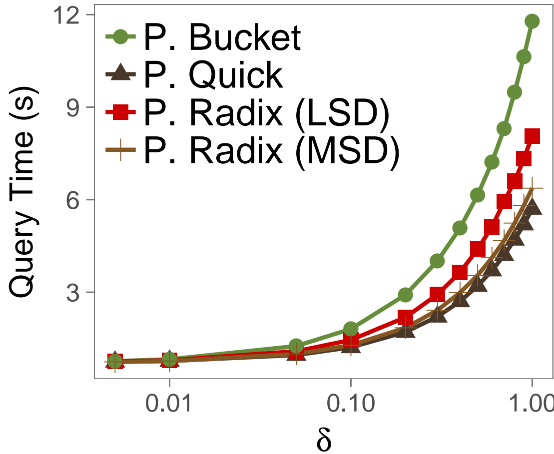


Figure 3-11: First Query.

Figure 3-11 shows the performance of the first query for varying values of δ. The first query's performance degrades as δ increases since each query does extra work proportional to δ. For every algorithm, however, the amount of extra work done differs.

We can see that Bucketsort is impacted the most by increasing δ. This is because determining which bucket an element falls into costs $O(log\, b)$ time, followed by a random write for inserting the element into the bucket. Radixsort, despite its similar nature to Bucketsort, is impacted much less heavily by an increased δ. This is because determining which bucket an element falls into costs constant $O(1)$ time. Quicksort

experiences the lowest impact from an increasing δ, as elements are always written to only two memory locations (the top and bottom of the array), the extra sequential writes are not very expensive.

Pay-Off.

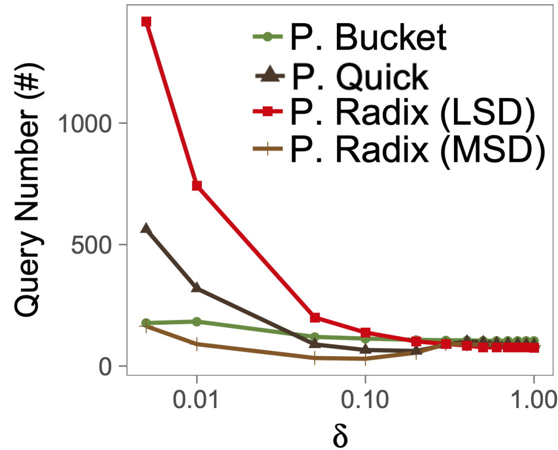


Figure 3-12: Pay-Off.

Figure 3-12 shows the number of queries required until the Progressive Indexing technique becomes worth the investment (i.e., the query number q for which $\sum_q t_{prog} \leq \sum_q t_{scan}$) for varying values of δ. We observe that with a very small δ, it takes many queries until the indexing pays off. While a small δ ensures low first query costs, it significantly limits the progress of index-creation per query, and consequently, the speed-up of query processing. With increasing δ, the number of queries required until pay-off quickly drops to a stable level.

We see that Radixsort (LSD) needs a very high amount of queries to pay-off for low values of δ. This is because the intermediate index cannot accelerate range queries until the index fully converges. When the value of δ is high, the index converges faster and can be utilized to answer range queries earlier. Quicksort also has a high time to pay-off with a low delta because the intermediate index can only be used to accelerate range queries that do not contain the pivots. Hence in the early stages of the index, the table often needs to be scanned. Bucketsort and Radixsort (MSD) do not suffer from these problems. Hence they pay-off fast even with lower values for δ.

Convergence.

The δ parameter affects the convergence speed towards a full index. When $\delta = 0$, the index will never converge, and a higher value for δ will cause the index to converge

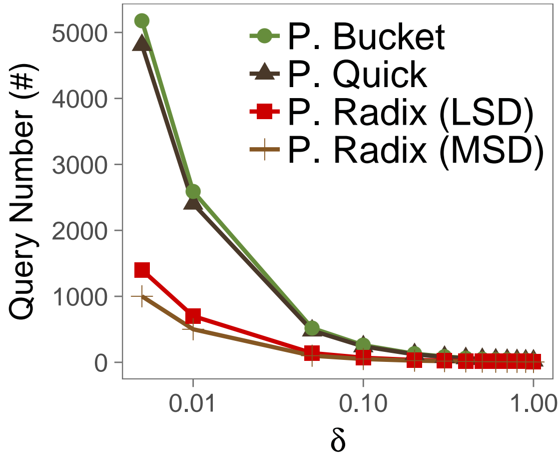


Figure 3-13: Convergence.

faster as more work is done per query on building the index.

Figure 3-13 shows the number of queries required until the index converges towards a full index. We see that Radixsort converges the fastest, even with a low δ. It is followed by Quicksort and then Bucketsort.

The reason Radixsort converges in so few iterations is because it uses radix partitioning, which means that after $\lceil log_2(n)/log_2(b) \rceil = \lceil log_2(10^9)/log_2(64) \rceil = 5$ partitioning rounds the index is fully converged. Bucketsort uses Quicksort pivoting, which requires more passes over the data.

Cumulative Time.

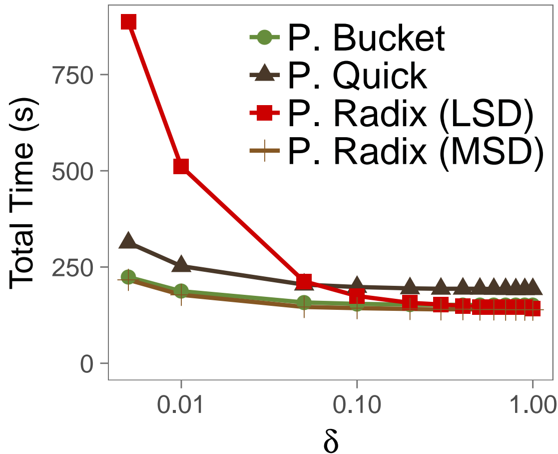


Figure 3-14: Total Time.

As we have seen before, a high value for δ means that more time is spent constructing the index, meaning that the index converges towards a full index faster. While earlier

queries take longer with a higher value of δ, subsequent queries take less time. Another interesting measurement is the cumulative time spent on answering a large number of queries. Does the increased investment in index creation earlier on pay off in the long run?

Figure 3-14 depicts the cumulative query cost. We can see that a higher value of δ leads to a lower cumulative time. Converging towards a full index requires the same amount of time spent constructing the index, regardless of the value of δ. However, when δ is higher, that work is spent earlier on (during fewer queries), and queries can benefit from the constructed index earlier.

Progressive Quicksort and Radixsort (LSD) perform poorly when the delta is low. For Quicksort, this is because it will take many queries to finish our pivoting in one element. While in Radixsort (LSD), the intermediate index that is created cannot be effectively used to answer range queries before it fully converges, meaning a long time until convergence results in poor cumulative time. Progressive Bucketsort and Radixsort (MSD) perform better than Progressive Quicksort for all values of δ, with Radixsort (MSD) slightly outperforming Bucketsort.

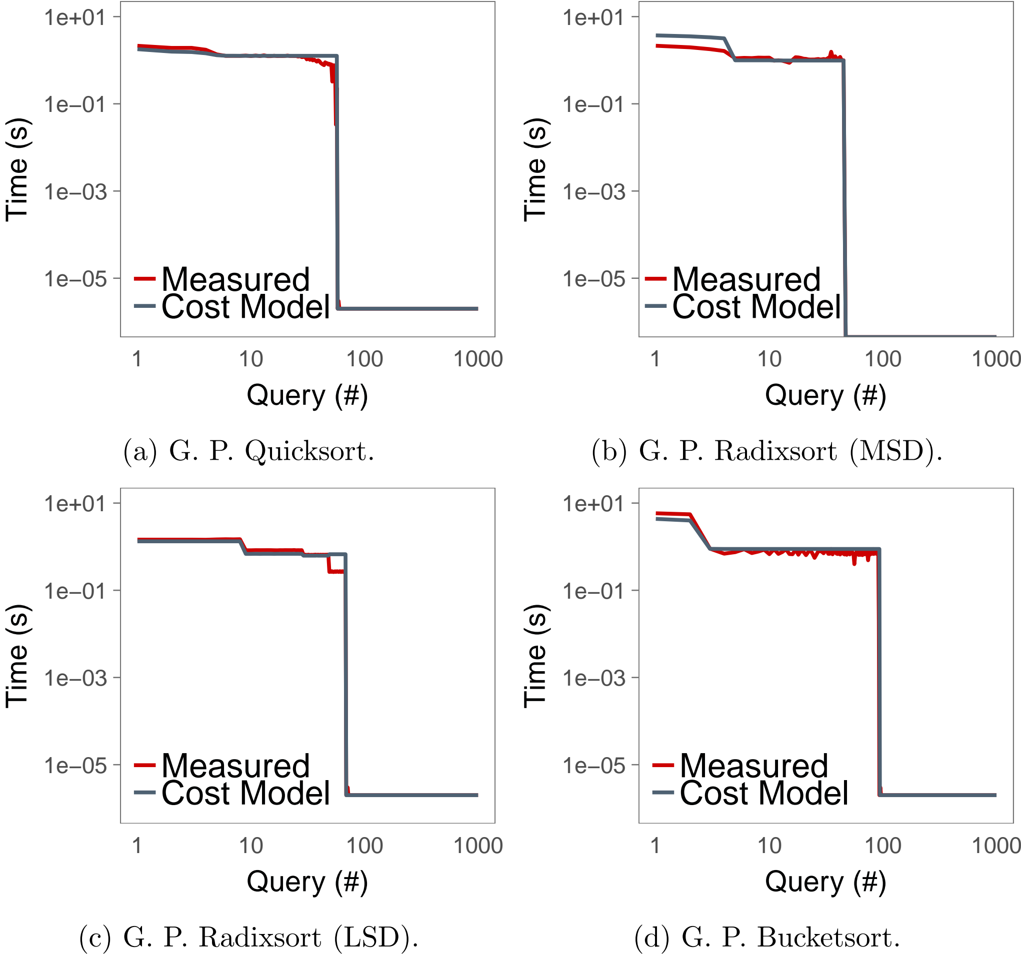
Another observation here is that the cumulative time converges rather quickly with an increasing delta. The cumulative time with $\delta = 0.25$ and $\delta = 1$ are almost identical for all algorithms, while the penalization of the initial query continues to increase significantly (recall Figure 3-11).

5.3 Cost Model Validation

For both the fixed indexing budget and the adaptive indexing budget of Greedy Progressive Indexing, we need the cost models presented in Section 4 to estimate the actual query processing and index creation costs. For the fixed indexing budget, we need the cost model to compute the initial value of δ based on the desired indexing budget. For the adaptive indexing budget, we need the cost model to adapt the value of δ for each query to the current minimum query cost.

In this set of experiments, we experimentally validate our cost models. To use the cost models in practice, we need to obtain values for all of the constants used, such as the scanning speed and the cost of a cache miss. Since these constants depend on the hardware, we perform these operations when the program starts up and measure how long it takes to perform these operations. The measured values are then used as the constants in our cost model.

Fixed Indexing Budget.



(a) G. P. Quicksort.

(b) G. P. Radixsort (MSD).

(c) G. P. Radixsort (LSD).

(d) G. P. Bucketsort.

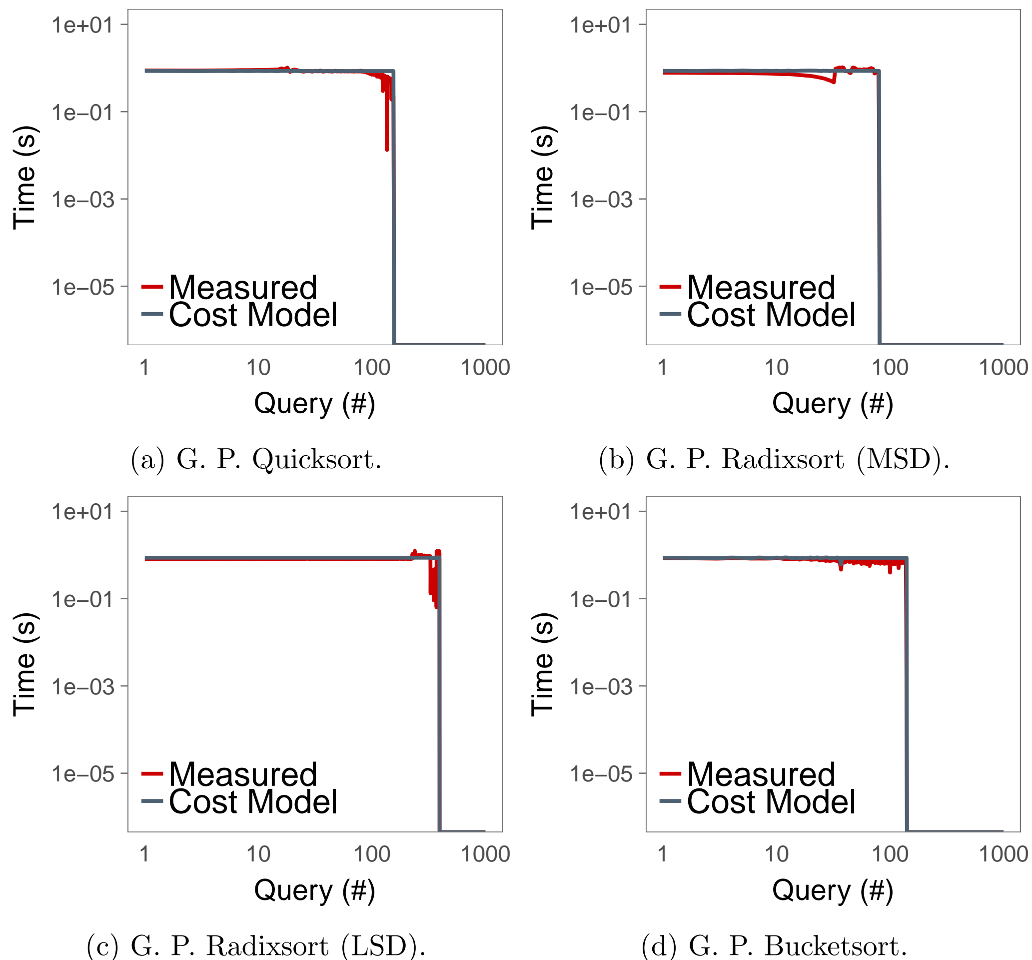
Figure 3-15: SkyServer Workload with Fixed Indexing Budget (all axes in log scale)

Before diving into the details of choosing a variable δ per query for the adaptive indexing budget, we first experimentally validate our cost models. We run the SkyServer benchmark with a constant $\delta = 0.25$ for the entire query sequence and compare the measured execution times with the times predicted by our cost models.

Figure 3-15 shows the results for all four Greedy Progressive Indexing techniques we propose. The graphs depict the individual phases of our algorithms (cf., Section 3) and show that significant improvements in query performance happen mainly with the transition from one phase to the next. Given that δ determines the fraction of data that is to be considered for index refinement with each query (rather than a fraction of the full scan cost), the different techniques depict different per query cost, depending on the respective index refinement operations performed as well as the efficiency of the respective partially built indexes. The graphs also show that our cost

models predict the actual costs well, accurately predicting each phase transition and the point when the full index has been finalized, and no further indexing is required.

adaptive indexing budget.



(a) G. P. Quicksort.

(b) G. P. Radixsort (MSD).

(c) G. P. Radixsort (LSD).

(d) G. P. Bucketsort.

Figure 3-16: SkyServer Workload with adaptive indexing budget (all axes in log scale)

With our cost models validated, we now run the SkyServer benchmark with all four Greedy Progressive Indexing techniques with the adaptive indexing budget. We select $t_{budget} = 0.2 * t_{scan}$, i.e., the indexing budget is selected as 20% of the full scan cost. Figure 3-16 depicts the results of this experiment for each of the algorithms. In all graphs, we observe that the total execution time stays close to constant at a high level, matching the given budget until the index is fully built, and no further refinement is required.

In Figure 3-16a, the measured and predicted time are shown for the Greedy Progressive Quicksort algorithm. Initially, the cost model accurately predicts the

performance of the algorithm. However, close to convergence, the cost model predicts a slightly higher execution time. As the pieces become smaller, they start fitting inside the CPU caches entirely, which results in faster swaps than predicted by our cost model.

In Figure 3-16b, the measured and predicted time are shown for the Greedy Progressive Radixsort (MSD) algorithm. In the initialization phase, the cost model matches the measured time initially, but the measured time slightly decreases below the cost model as the initialization progresses. This is because the data distribution is relatively skewed, which results in the same buckets being scanned for every query, which will then be cache resident and faster than predicted. In the refinement phase, there are some minor deviations from the cost model caused by smaller radix partitions fitting in CPU caches, which our cost model does not accurately predict.

In Figure 3-16c, the measured and predicted time are shown for the Greedy Progressive Radixsort (LSD) algorithm. The cost model accurately predicts the performance of the initialization and refinement phases of the algorithm but results in several spikes later in the refinement phase. These spikes occur because the workload we are using consists of very wide range queries. These range queries can only take advantage of the LSD index depending on the exact range queries issued. Thus, certain queries can be answered much faster using the index, whereas others cannot use the index at all. As our cost model is pessimistic, this results in the measured time being faster than the predicted time.

In Figure 3-16d, the measured and predicted time are shown for the Greedy Progressive Bucketsort algorithm. In the initialization phase, the cost model closely matches the measured time. After it, Greedy Progressive Quicksort is used to merge the different buckets into a single sorted array. The different iterations of Greedy Progressive Quicksort each have small downwards spikes when the pieces start fitting inside the CPU caches.

5.4 Interactivity Threshold

In the previous workload, we have shown our cost models' effectiveness at staying on a specific interactivity threshold. In this experiment, we want to show how the algorithms perform at different interactivity thresholds based on the full scan cost. In this experiment, we show three different scenarios: (1) the interactivity threshold is below the full scan cost, (2) the interactivity threshold is above the full scan cost, and (3) the interactivity threshold decreases with the number of queries issued.

Threshold Below Full Scan Cost.

In the first scenario, the initial runs will always be above the interactivity threshold as even the full scan cannot reach it. In this scenario, we start by setting δ to 0.25. In Section 5.2 we determined this provides a fast convergence rate while not heavily penalizing the initial queries. After the index has reached the state where it can answer queries below the interactivity threshold, δ is set such that the query cost stays on the interactivity threshold until convergence.

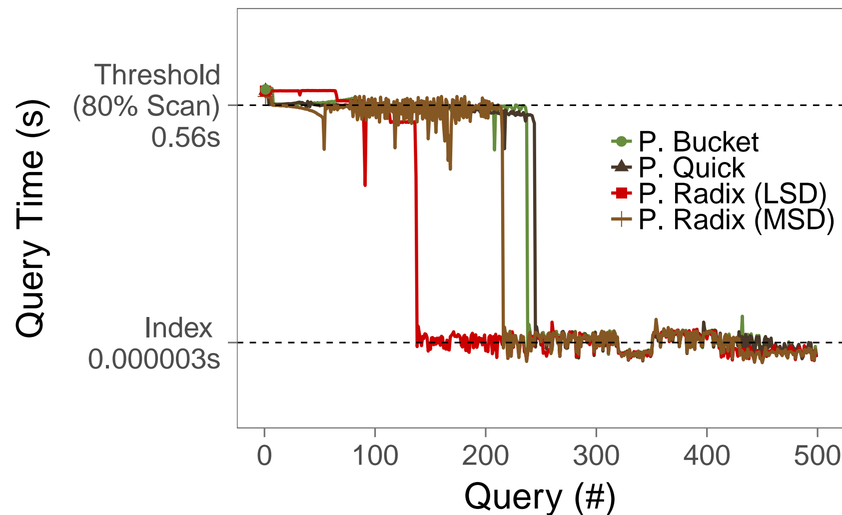


Figure 3-17: Threshold of 80% Scan Cost (Y-Axis in log scale)

Figure 3-17 shows the results for this experiment. We can see that all queries start above the interactivity threshold, after which they gradually move towards it. The Greedy Progressive Quicksort and Radixsort (MSD) quickly reach the interactivity threshold. The Radixsort (LSD) takes the longest to reach it. This is because the wide range queries cannot take advantage of the LSD radix index structure to speed up answering the queries. However, because it stays on the δ of 0.25 the longest, i.e., it performs more indexing work with more initial queries, it does converge the fastest.

Figure 3-18 shows the time spent on indexing versus the time spent on query processing for the Greedy Progressive Quicksort in this scenario. At the start, a significant amount of time is spent on indexing as the interactivity threshold cannot be reached yet. After the index is sufficiently converged, the interactivity threshold can be reached, and the fixed $\delta = 0.25$ is replaced by a variable per-query δ as discussed in Section 5.2, which is initially rather small given that the time budget between query processing cost and interactivity threshold is small. As more data gets indexed, the query processing cost gradually decreases. Consequently, δ is gradually

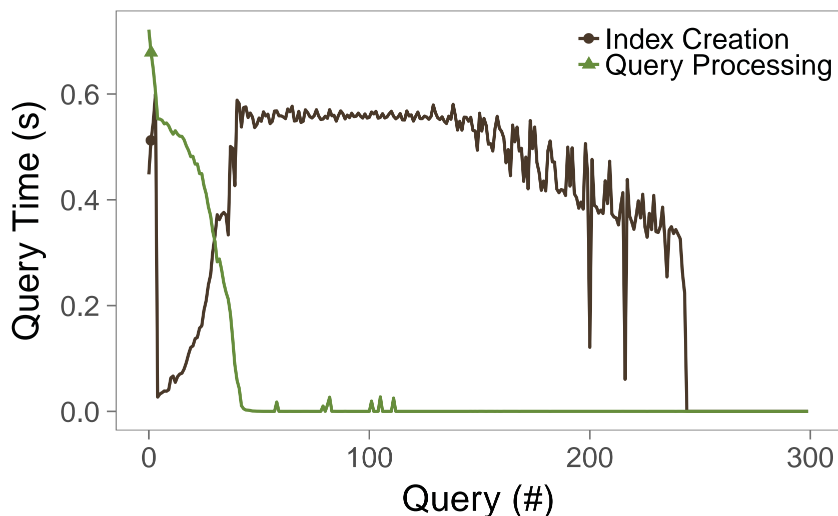


Figure 3-18: Progressive Quicksort - Query Processing vs. Index Creation

increased, allowing to spend more time on index creation per query until the index fully converges.

Threshold Above Full Scan Cost.

In the second scenario, all the greedy algorithms can stay on the interactivity threshold above the full scan cost. The only difference is the time until convergence for each of the algorithms. These differ based on how much extra time we can spend on index creation, which depends on how much the interactivity threshold is above the full scan cost. For this reason, we performed two separate experiments, one where the interactivity threshold is $1.2x$ the full scan cost and one where it is $2x$ the full scan cost.

Figure 3-19 shows the experiment results where the threshold is $1.2x$ the scan cost. In this experiment, the Radixsort (MSD) converges the fastest, and the Radixsort (LSD) converges the slowest. This is because the intermediate index created by the Radixsort (LSD) cannot be effectively used to speed up the range queries, and the δ will stay at a fixed low number until convergence. As the interactivity threshold is so close to a full scan, this value will be very low. The other indexes can use the intermediate index to speed up the query processing, resulting in an increasing δ that improves convergence time.

Figure 3-20 shows the results of the experiment where the threshold is $2x$ the scan cost. In this experiment, the Radixsort (LSD) converges the fastest. While the intermediate index still cannot speed up the query, as the full scan only takes

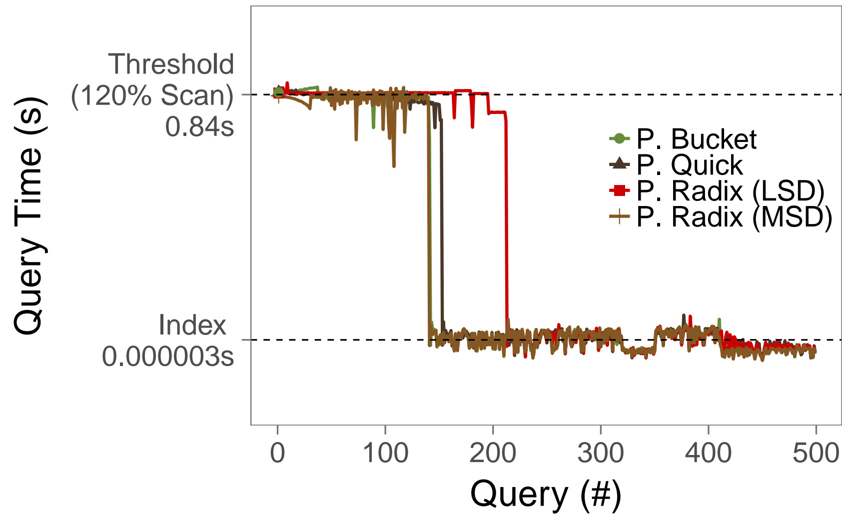


Figure 3-19: Threshold of 120% Scan Cost (Y-Axis in log scale)

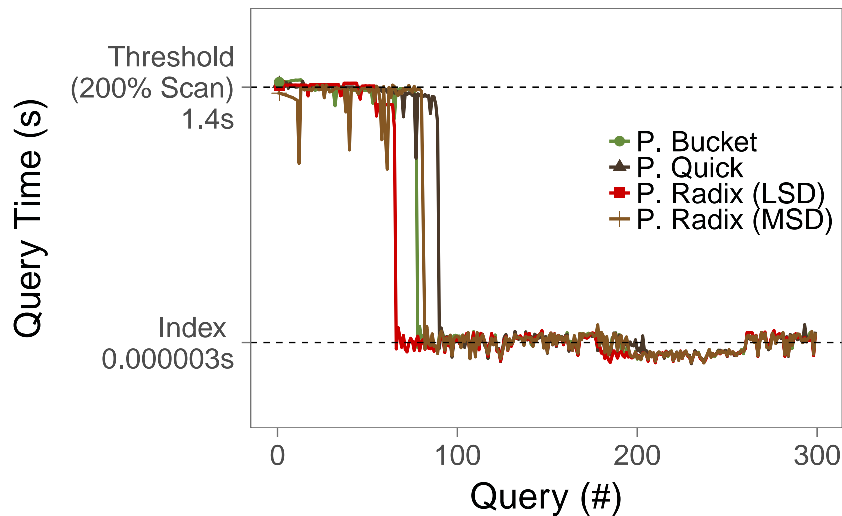


Figure 3-20: Threshold of 200% Scan Cost (Y-Axis in log scale)

up half the interactivity threshold, the amount of time spent on index refinement is significantly higher than in the previous experiment for all algorithms. As the Progressive Radixsort (LSD) has the fastest convergence, as shown in Figure 3-13, it is now the fastest converging algorithm.

5.5 Varying Interactivity

So far, we have used the same fixed interactivity threshold for all queries. Fully exploiting the time budget between this threshold and pure query processing cost for index creation ensures faster convergence towards a full index. However, it also results

in a rather discrete behavior: All initial queries (including index refinement) take as long as allowed by the interactivity threshold. Once the full index is entirely built, query times abruptly drop to the optimal times using the index. This behavior might not always be desirable. Instead, a more gradual convergence of the query execution times from the given interactivity threshold to the optimal case might be preferred, possibly at the expense of slightly slower convergence.

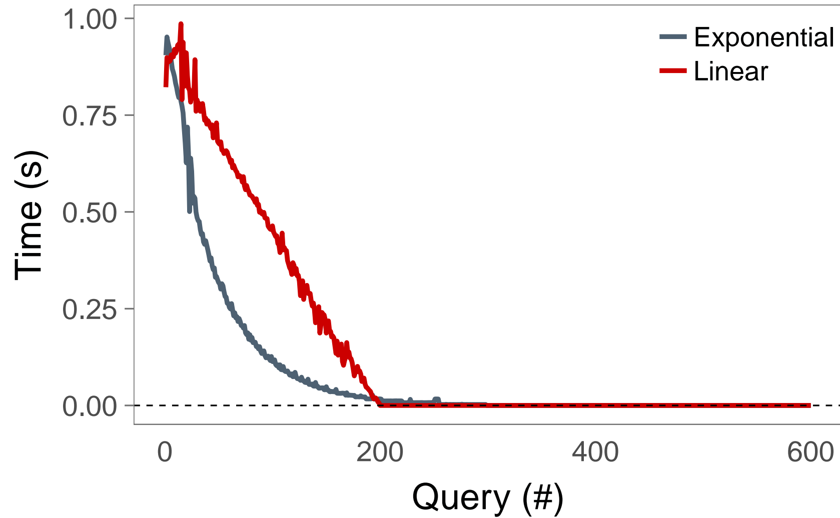


Figure 3-21: Exponential and Linear Decay

We can mimic such behavior by monotonously decreasing the interactivity threshold with the query sequence, ensuring that at any time, the interactivity threshold remains high enough so that the queries can be completed within that threshold. Again, using our cost model, we can automatically determine the respective values for δ. We perform two experiments using linear and exponential decay formulas to model the decreasing the interactivity threshold depicted in Figure 3-21.

For the linear decay, we set our formula as $t_q = I - r * q$ where I is the initial interactivity threshold, t_q is the total time spent on query number q, and r is the decreasing ratio. We use the following formula to calculate the decay ratio $r = -\frac{t_{FI}-I}{n}$ where t_{FI} is the estimated full index time and n is the amount of queries until convergence to time t_{FI}. For this experiment, we use Progressive Quicksort with $n = 200$ and the initial interactivity threshold set to $1.2x$ the full scan time. We can see that we can gradually push down the execution time as the index converges by gradually decreasing the interactivity threshold.

For the exponential decay, we use the exponential decay formula $t_q = I(1-r)^q$. To determine r, we use the following formula $r = 1 - \sqrt[n]{\frac{t_{FI}}{I}}$. For this experiment, we use Progressive Quicksort with $n = 300$ and the initial interactivity threshold set to $1.2x$

the full scan time. Like the linear decay, we can see that the measured time closely follows the interactivity threshold.

5.6 Adaptive Indexing Comparison

In this section, we will be comparing the greedy Progressive Indexing techniques with existing Adaptive Indexing techniques. In particular, we focus on Standard Cracking (STD), Stochastic Cracking (STC), progressive Stochastic Cracking (PSTC), Coarse Granular Index (CGI), and Adaptive Adaptive Indexing (AA).

The implementations for the Full Index, Standard Cracking, Stochastic Cracking, and Coarse Granular Index were inspired by the work done in Schuhknecht et al. [50][4]. The implementation for Progressive Stochastic Cracking was inspired by the work done in Halim et al. [26][5]. Progressive Stochastic Cracking is run with the allowed swaps set to 10% of the base column. The implementation for the Adaptive Adaptive Indexing algorithm has been provided to us by the authors of the Adaptive Adaptive Indexing work [49], and we use the manual configuration suggested in their paper.

We compare all the Progressive Indexing techniques that we have introduced in this work: Greedy Progressive Quicksort (PQ), Greedy Progressive Bucketsort (PB), Greedy Progressive Radixsort LSD (PLSD), and Greedy Progressive Radixsort MSD (PMSD). For each of the techniques, we use an adaptive indexing budget where we set $t_{budget} = 0.2 * t_{scan}$, i.e., the cost of each query will be equivalent to $1.2 * t_{scan}$ until convergence.

For reference, we also include the timing results when only performing full scans on the data (FS) and when constructing a full index immediately on the first query (FI). The full scan implementation uses predication to avoid branches, and the full index bulk loads the data into a B+-tree, after which the B+-tree is used to answer subsequent queries.

Metrics. The metrics that we are interested in are the time taken for the first query, the number of queries required until convergence, the robustness of each of the algorithms, and the cumulative response time. The robustness we compute by taking the variance of the first 100 query times.

Table 3.2: SkyServer Results

Index	First Q	Convergence	Robustness	Cumulative
FS	0.75	x	0	118743.7
FI	34.10	1	x	121.4
STD	5.26	x	0.290	1082.2
STC	4.99	x	0.250	245.6
PSTC	4.89	x	0.240	254.5
CGI	5.71	x	0.320	1008.9
AA	8.50	x	0.800	188.4
PQ	0.90	150	0.002	202.9
PMSD	0.90	119	0.030	157.5
PLSD	0.81	368	3.4e-05	377.4
PB	0.83	138	0.009	166.4

SkyServer Workload

In this part of the experiments section, we execute the full SkyServer workload using different indexing techniques. The results for each of the indexing techniques are shown in Table 3.2. The algorithms have been divided into three sections: the baseline, the Adaptive Indexing techniques, and the Progressive Indexing techniques.

The results for the baseline techniques are not very surprising. The full scan method is the most robust, as we use predication, and no index is constructed. The cost of each query is identical. The full scan method is also the cheapest method for the first query's cost as no time is spent on indexing at all. The full scan, however, takes significantly longer to answer the full workload than the other methods. Answering the full workload takes almost 30 hours, whereas all the other techniques finish the entire workload under 20 minutes. The full index lies at the other extreme. It takes $50x$ longer to answer the first query while the index is being constructed. However, it has the lowest cumulative time as the index can quickly answer all of the remaining queries.

For the Adaptive Indexing techniques, we can see that their first query cost is significantly lower than that of a full index but still significantly higher than that of a full scan. Each of the Adaptive Indexing methods performs a significant amount of work copying the data and cracking the index on the first query, resulting in a very high cost for the first query. They do achieve a significantly faster cumulative time than the full scans. However, in sum, they take longer than the full index to answer the workload. Standard Cracking and Coarse Granular Indexing perform particularly

[4] https://infosys.uni-saarland.de/publications/uncracked_pieces_sourcecode.zip
[5] https://github.com/felix-halim/scrack

poorly because of the workload's sequential nature, as shown in Figure 3-9. Stochastic Cracking and Adaptive Indexing perform better as they do not choose the pivots based on the query predicates. Adaptive Adaptive Indexing has the best cumulative performance, consistent with the results in Schuhknecht et al. [49].

The Progressive Indexing methods all have approximately the same cost for the first query, which is $1.2x$ the scan cost. This is by design as we set the indexing budget $t_{budget} = 0.2 * t_{scan}$ for each of the algorithms. The main difference between the algorithms is the robustness and the time until convergence. As we are executing range queries, the Radixsort LSD performs the worst. The LSD partitioning cannot help answer the range queries, and hence, the intermediate index does not speed up the workload before convergence. Radixsort MSD performs the best, as the data set is rather uniformly distributed. The radix partitioning works to efficiently create a partitioning of the data, which can be immediately utilized to speed up subsequent queries. For each of the Progressive Indexing methods, we see that they converge relatively early in the workload. As we have set every query to take $1.2 * t_{scan}$ until convergence, a significant amount of time can be spent on constructing the index for each query, especially in later queries when the intermediate index can already be used to obtain the answer efficiently. We also note that the Progressive Indexing methods have a significantly higher robustness score than the Adaptive Indexing methods. Progressive Indexing presents up to 4 orders of magnitude lower query variance when compared to the Adaptive Indexing techniques. This is achieved by our cost model balancing the per query execution cost to be (almost) the same until convergence, while Adaptive Indexing suffers from many performance spikes.

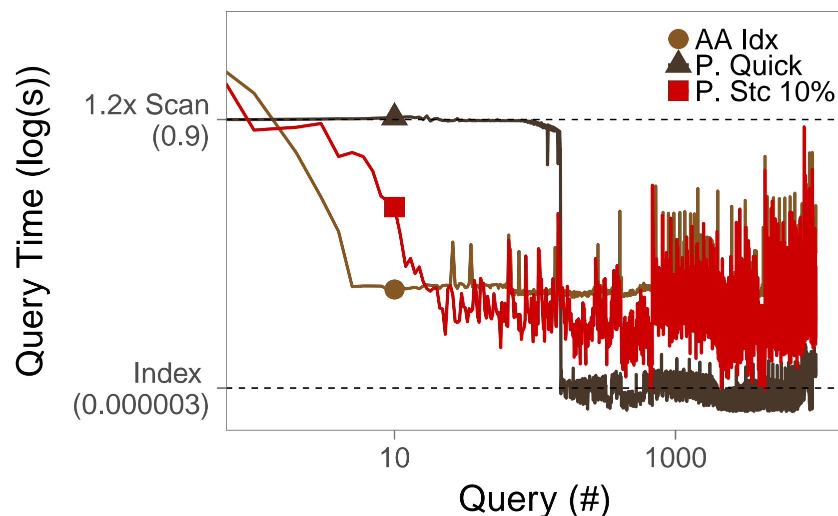


Figure 3-22: Progressive Quicksort vs Adaptive Indexing. (all axes in log scale)

The execution time for each of the queries in the SkyServer workload is shown in Figure 3-22. For clarity, we focus on the best Adaptive Indexing methods (Adaptive Adaptive Indexing in terms of cumulative time, and Progressive Stochastic 10% in terms of first query cost and robustness) and Progressive Quicksort. We can see that both the Adaptive Indexing methods start with a significantly higher first query cost and then fall quickly. Neither of them sufficiently converges, however, and both continue to have many performance spikes. On the other hand, Progressive Quicksort starts at the specified budget and maintains that query cost until convergence, after which the cost drops to the cost of a full index.

Synthetic Workloads

In this part of our experiments, we execute all synthetic workloads described in Section 5.1. All results are presented in tables. Each table is divided into four parts, each representing one set of experiments. The first three are on data with 10^8 elements and use random distribution, skewed distribution, and only point queries, respectively. The final one is on 10^9 elements on random distribution. With the exception of point queries and the ZoomIn and SeqZoomIn workloads, all queries have 0.1 selectivities. From the Adaptive Indexing techniques, Adaptive Adaptive Indexing presents the best cumulative time. Hence we select it for comparison. As previously, we set the indexing budget $t_{budget} = 0.2 * t_{scan}$ for each Progressive Indexing algorithm.

Table 3.3 depicts the cost of the first query for all algorithms. All Progressive Indexing algorithms present a similar first query cost, which accounts for approximately $1.2x$ the scan cost, as chosen in our setup. Adaptive Indexing has a higher cost due to the complete copy of the data and its full partition step in the first query. In general, Progressive Indexing has one order of magnitude faster first query cost than Adaptive Indexing.

Table 3.4 depicts the cumulative time of fully executing each workload. We can see that Progressive Indexing outperforms Adaptive Indexing in most workloads under uniform random data, except for the skewed and the periodic workload. This comes with no surprise since Adaptive Indexing techniques have been designed to refine, and boost access, to frequently accessed parts of the data. From the progressive algorithms, radixsort (MSD) is the fastest since radixsort can outperform other techniques under randomly distributed data.

For the skewed distribution, Adaptive Indexing outperforms Progressive Indexing in almost all workloads due to its refinement strategy. However, Progressive Indexing outperforms Adaptive Indexing for ZoomIn/Out workloads since each query accesses a

Table 3.3: First query cost

	Workload	PQ	PB	PLSD	PMSD	AA
Uniform Random	SeqOver	0.15	0.15	0.14	0.14	1.4
	ZoomOutAlt	0.15	0.15	0.14	0.14	1.4
	Skew	0.15	0.15	0.14	0.14	1.4
	Random	0.15	0.15	0.14	0.14	1.4
	SeqZoomIn	0.15	0.15	0.14	0.14	1.4
	Periodic	0.15	0.15	0.14	0.14	1.4
	ZoomInAlt	0.15	0.15	0.14	0.14	1.4
	ZoomIn	0.15	0.15	0.14	0.14	1.4
Skewed	SeqOver	0.15	0.15	0.14	0.14	1.5
	ZoomOutAlt	0.15	0.15	0.14	0.13	1.5
	Skew	0.15	0.15	0.14	0.13	1.5
	Random	0.15	0.15	0.13	0.13	1.5
	SeqZoomIn	0.15	0.15	0.14	0.13	1.5
	Periodic	0.15	0.15	0.14	0.13	1.5
	ZoomInAlt	0.15	0.15	0.14	0.14	1.5
	ZoomIn	0.15	0.15	0.14	0.14	1.5
Point Query	SeqOver	0.15	0.15	0.21	0.14	1.4
	ZoomOutAlt	0.15	0.15	0.21	0.14	1.4
	Skew	0.15	0.15	0.21	0.14	1.4
	Random	0.15	0.15	0.21	0.14	1.4
	Periodic	0.15	0.15	0.21	0.14	1.4
	ZoomInAlt	0.15	0.15	0.21	0.14	1.4
10^9	SeqOver	1.5	1.5	1.4	1.7	13.9
	Skew	1.5	1.5	1.4	1.7	13.8
	Random	1.5	1.5	1.4	1.7	25.4

different partition in different boundaries of the data, which leads to Adaptive Indexing accessing large unrefined pieces in the initial queries. From the progressive algorithms, bucketsort presents the fastest times since it generates equal-sized partitions for skewed data distributions.

For point queries, radixsort (LSD) outperforms all algorithms in all workloads since its intermediate index can be used early on to accelerate point queries.

Finally, for the 10^9 data size, Progressive Indexing manages to outperform adaptive indexing even for the skewed workload. The key difference here is that the chunks of unrefined data are bigger, and Progressive Indexing actually spends the time fully converging them into small pieces. At the same time, Adaptive Indexing must manage larger pieces of data.

Table 3.5 presents the robustness of the indexing algorithms. Progressive Indexing presents up to four orders of magnitudes less variance than Adaptive Indexing. This is due to the design characteristic of Progressive Indexing to inflict a controlled indexing

Table 3.4: Cumulative Time

	Workload	PQ	PB	PLSD	PMSD	AA
Uniform Random	SeqOver	19.0	17.9	48.2	16.2	20.7
	ZoomOutAlt	20.7	28.3	59.5	26.7	22.1
	Skew	18.8	17.7	48.1	15.9	10.1
	Random	24.7	22.8	53.1	21.1	29.1
	SeqZoomIn	22.0	20.9	53.5	19.3	21.1
	Periodic	23.3	22.0	63.9	20.4	18.4
	ZoomInAlt	20.8	23.3	54.2	21.6	21.7
	ZoomIn	167.0	165.0	210.0	164.0	277.0
Skewed	SeqOver	21.8	30.0	59.7	21.7	17.5
	ZoomOutAlt	21.5	30.2	64.4	63.7	41.1
	Skew	17.4	15.3	45.5	17.3	5.7
	Random	24.0	21.6	51.5	23.8	23.9
	SeqZoomIn	23.3	21.2	52.6	23.1	18.3
	Periodic	23.3	21.3	64.2	23.3	17.0
	ZoomInAlt	22.2	25.1	54.8	21.8	33.5
	ZoomIn	938.0	919.0	934.0	917.0	1655.0
Point Query	SeqOver	16.7	15.7	13.2	14.0	15.1
	ZoomOutAlt	17.7	15.8	13.0	14.0	15.5
	Skew	16.6	15.5	12.7	13.7	5.6
	Random	18.4	16.5	13.6	14.7	14.4
	Periodic	16.8	15.7	13.0	14.3	5.7
	ZoomInAlt	17.7	15.9	13.2	14.1	15.2
10^9	SeqOver	516	493	924	480	653
	Skew	538	513	885	487	582
	Random	773	718	1579	692	1104

penalty. For uniform random and skewed distributions, radixsort LSD presents the least variance. This is due to the cost model noticing that the intermediate index created by LSD cannot be used to boost query access, hence knowing the precise cost of executing the query (i.e., a full scan cost). However, for point queries, the intermediate index from LSD can already be used, which reduces the cost model accuracy.

6 Summary

This chapter introduces Progressive Indexing, a novel incremental indexing technique that offers robust and predictable query performance under different workloads. Progressive techniques perform indexing within an interactivity threshold and provide a balance between fast convergence towards a full index together with a small performance penalty for the initial queries. We propose four different Progressive Indexing

Table 3.5: Robustness

	Workload	PQ	PB	PLSD	PMSD	AA
Uniform Random	SeqOver	2.4e-04	5.8e-04	2.2e-05	2.1e-04	0.02
	ZoomOutAlt	1.7e-04	6.0e-04	2.1e-05	2.1e-04	0.02
	Skew	2.5e-04	6.2e-04	2.9e-05	2.3e-04	0.02
	Random	2.1e-04	6.5e-04	2.3e-05	2.0e-04	0.02
	SeqZoomIn	2.3e-04	5.5e-04	2.6e-05	2.1e-04	0.02
	Periodic	2.4e-04	6.6e-04	1.9e-05	2.1e-04	0.02
	ZoomInAlt	2.4e-04	5.4e-04	2.2e-05	2.1e-04	0.02
	ZoomIn	2.3e-04	3.8e-04	3.1e-05	1.4e-04	0.02
Skewed	SeqOver	3.7e-04	7.5e-04	1.6e-05	2.5e-03	0.03
	ZoomOutAlt	3.1e-04	7.6e-04	1.4e-05	2.7e-04	0.03
	Skew	3.5e-04	7.9e-04	1.4e-05	2.5e-03	0.03
	Random	3.4e-04	7.8e-04	1.9e-05	2.5e-03	0.03
	SeqZoomIn	3.6e-04	8.5e-04	1.4e-05	2.5e-03	0.03
	Periodic	3.2e-04	8.2e-04	1.5e-05	2.4e-03	0.03
	ZoomInAlt	3.4e-04	7.5e-04	1.4e-05	2.5e-03	0.02
	ZoomIn	1.9e-05	2.3e-04	1.4e-05	1.4e-03	0.02
Point Query	SeqOver	2.4e-04	7.0e-04	1.5e-03	2.2e-04	0.02
	ZoomOutAlt	1.8e-04	6.3e-04	1.6e-03	2.1e-04	0.02
	Skew	2.6e-04	6.8e-04	1.6e-03	2.3e-04	0.02
	Random	2.2e-04	6.6e-04	1.6e-03	2.5e-04	0.02
	Periodic	2.2e-04	6.8e-04	1.1e-03	2.1e-04	0.02
	ZoomInAlt	2.3e-04	6.8e-04	1.5e-03	3.3e-04	0.02
10^9	SeqOver	0.02	0.03	2.8e-04	0.04	2.1
	Skew	8.1e-03	0.03	1.0e-04	0.03	2.1
	Random	0.01	0.03	2.4e-04	0.02	7.0

techniques and develop cost models for all of them that allow automatic tuning of the budget. We show how they perform with both real and synthetic workloads and compare their performance against Adaptive Indexing techniques. Based on

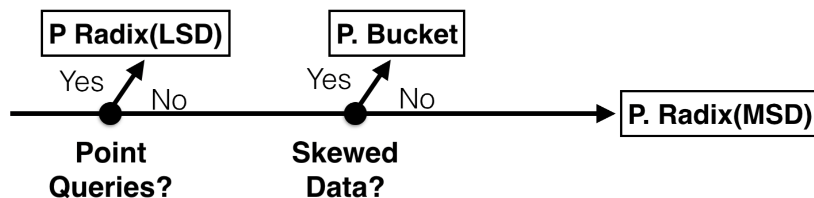


Figure 3-23: Progressive Indexing Decision Tree.

each algorithm's main characteristics and the results of our experimental evaluation, we conclude our work with the decision tree shown in Figure 3-23, which provides recommendations on which technique to use in different situations. In this chapter, we only present an algorithm to index unidimensional columns. However, queries frequently have filters on multiple columns. In the next chapter, we will describe how

to create a Progressive Index that indexes multiple dimensions.

CHAPTER 4

Multidimensional Progressive Indexing

1 Introduction

As seen in the previous chapter, techniques like Adaptive Indexing [36, 50] and Progressive Indexing (Chapter 3) aim to alleviate the index construction issue on exploratory workloads by creating partial unidimensional indexes as a result of query processing. In this way, indexes are automatically created without any human intervention and incrementally refined towards a full index, the more the data is accessed. However, these techniques have very limited use on a broad group of data sets since they only target unidimensional workloads. For instance, the 1000 genomes project [16] has human genetic information, the Power data set[1] that contains sensor information from a manufacturing installation, and the SkyServer data set [56] which maps the universe, are some of many examples that perform multidimensional filtering.

Pavlovic et al. [45] published a study on multidimensional adaptive indexes, initially testing a Space-Filling Curve strategy, where multiple dimensions are mapped to one dimension. They used unidimensional adaptive indexing techniques on top of the created map. However, the first queries' indexing burden was too high, making this approach unfeasible for interactive times. They later propose the QUASII index, a d-level hierarchical structure that similarly partitions the data as the coarse granular index strategy [50]. When accessing one piece, the data is continuously refined until all pieces are smaller than a given size threshold. This strategy is much more efficient

[1] https://debs.org/grand-challenges/2012/

in smearing out the cost of index creation than the Space-Filling Curve Adaptive Indexing. However, it results in two highly undesirable characteristics for exploratory workloads. (1) Due to the continuous piece refinement, it heavily penalizes queries when they first access one piece; (2) since the index prioritizes an aggressive refinement only on areas targeted by the executing query, it is not robust against changes in the access pattern, resulting in performance spikes if the workload suddenly accesses a previously unrefined piece.

This chapter introduces two novel algorithms to tackle the problem of multidimensional adaptive indexing under exploratory data analysis. (a) The *Progressive KD-Tree*, inspired by fixed-delta progressive indexing, introduces a per-query indexing budget that remains constant during query execution. Hence, a user-controlled amount of indexing is done per query. (b) The *Greedy Progressive KD-Tree* uses a cost model to automatically adapt the indexing budget to keep the per-query cost constant until full index convergence, achieving a low variance per-query.

1.1 Contributions

The main contributions of this chapter are:

- We introduce a new progressive indexing approach for multidimensional workloads named *Progressive KD-Tree*.
- We present a cost model for our Progressive KD-Tree to enable an adaptive indexing budget.
- We experimentally verify that our techniques improve total execution time, initial query cost, robustness, and convergence compared with the state-of-the-art.
- We provide an Open-Source implementation[2] of all techniques discussed in this chapter.

1.2 Outline

This chapter is organized as follows. In Section 2, we investigate related research that has been performed on multidimensional indexes. In Section 3, we describe our novel Multidimensional Progressive Indexes. In Section 4, we perform a quantitative assessment of each of the novel methods we introduce, and we compare them with

[2] Our implementations and benchmarks are available at `https://github.com/pdet/MultidimensionalAdaptiveIndexing` and `https://zenodo.org/record/3835562`

the state-of-the-art on multidimensional adaptive indexing under three real workloads and eight synthetic workloads. Finally, in Section 5, we draw our conclusions.

2 Related Work

In the previous chapter we discussed how the selection of which indexes to create has been a long-standing problem in database automatical physical design. However, the selection of the indexes is just part of the problem. Another equally important problem is selecting which data-structure to use since each structure is catered to different workload patterns and data distributions. Multidimensional access methods can be distinguished between point access methods (PAMs) and spatial access methods (SAMs) [18]. Typically, PAMs aim at databases storing only points with support to spatial searches on them, like KD-Trees, PH-Tree, and flat structures. The term point refers to both locations in space and point objects stored in the database. SAMs, like R-Trees and Z-Ordering, aim at extended objects (e.g., polygons in geographic databases) while, like PAMs, also storing points [55].

In this section, we briefly discuss the state-of-the-art multidimensional index structures.

2.1 Multidimensional Data Structures

R-Tree [24]

The R-Tree is an N-ary multidimensional tree that generalizes the B-Tree. Nodes represent rectangles that bound the insertion points of data (i.e., coverage), and different rectangles may overlap data. Like B-Trees, the insertions and deletions require splitting and merging nodes to preserve height-balance with all leaves at the same depth. The internal nodes keep a way of identifying a child node and representing the boundaries of the entries in the child nodes, while the external nodes store the data. The R-Tree has a variant, the R*-Tree [5], for read-mostly workloads that balances the rectangle coverage and reduces overlapping.

VA File [61]

The VA File is a flat structure that divides an m-dimensional space in 2^b rectangular cells. Users assign b bits to be distributed over the m dimensions. A unique bit-string of length b is set for each cell, and data objects use a hash method to find the spacial position to each value (i.e., approximation by the bit-string).

KD-Tree [7]

The KD-Tree is a multidimensional binary search tree, where k is the number of dimensions of the search space that are switched between tree levels. The performance of KD-Trees is of great advantage as searches, insertions, and deletions of random nodes present logarithmic complexity and search of t tuples present sub-linear complexity. The nodes of the tree are insertion points. Therefore, the order of insertion shapes the tree structure but increases the complexity of maintenance when tree re-balancing is needed after deletion.

PH-Tree [63]

The PH-Tree implements a bit-string prefix sharing tree to reduce the space requirements compared to single key storage. The bit-string representation is used to navigate the dimensions in a Quadtree, where the first bit of the index entry indicates the position in the search space.

This approach is advantageous in data sets where data points are not evenly spread and share many prefixes. Otherwise, spread out data with large number of dimensions increases the number of nodes and the depth of the prefix tree, which also increases the space requirements and the lookup time.

Flood [42]

Flood is a multidimensional learned index. The learning algorithm's goal is to help to tweak performance parameters of the index, like the layout of the index by choosing between a grid of cells or columns (in a 2-D representation), the size of each cell, and the sort order of each cell or column.

Discussion.

To compare these index structures, we must put them in the context of the data exploration scenario. Although Flood has a significant advantage of finding an efficient setup by searching the parameters' space, it is not a good fit for our types of workloads since it requires a considerable amount of time to be invested in model training (i.e., index creation). PH-Trees present efficient lookups, but they are catered to data sets where data points are not evenly spread and share many prefixes. Finally, KD-Trees, VA Files, and R*-Trees have been thoroughly examined, in the main memory context, by Sprenger et al. [55]. The work concludes that the KD-Trees outperform R*-Trees and VA Files due to its point storage design. VA-Files have even a more

significant disadvantage for shifting access patterns, common in exploratory data analysis, since it is a non-adaptive structure with a static number of bits assigned per dimension. Considering each technique's main drawbacks and advantages, we decided to use a KD-Tree as our multidimensional index of choice for exploratory data analysis, as a full index baseline and the index structure that holds the data for our progressive solution. In summary, the reasons are its robust performance against shifting workloads, different from VA Files and PH Trees, the higher performance when compared to R*-Trees, and low index creation cost compared to Flood.

2.2 Adaptive/Progressive Index

In this section, we discuss the state-of-the-art adaptive indexing techniques that produce multidimensional indexes.

Space Filling Curve Cracking [45]

Space Filling Curve Cracking uses a space-filling curve technique that preserves proximity (e.g., Z-Order, Hilbert Curve) to map multiple dimensions into one dimension. This additional step enables the use of unidimensional adaptive/progressive indexing techniques. Later on, queries also must be translated to this unidimensional mapping.

QUASII [45]

Following the adaptive indexing philosophy, QUASII incrementally builds a multidimensional index prioritizing refinement on queried pieces. One significant difference compared to standard adaptive indexing techniques is that QUASII has a more aggressive refinement behavior. When accessing a piece, it recursively refines it until its size drops below a *size_threshold*. QUASII pays higher refinement costs when a piece is accessed the first time to yield fast query response time when frequently accessing refined pieces.

Adaptive KD-Tree [43]

The *Adaptive KD-Tree* is a multidimensional adaptive indexing technique that follows the same principles as Adaptive Indexing [36]: (1) It uses the query predicate as hints as to what pieces of the data should be indexed, and (2) only indexes the necessary pieces to answer the current query. Our index has two main canonical phases. The *initialization* phase only happens when the first query selecting a group of non-indexed

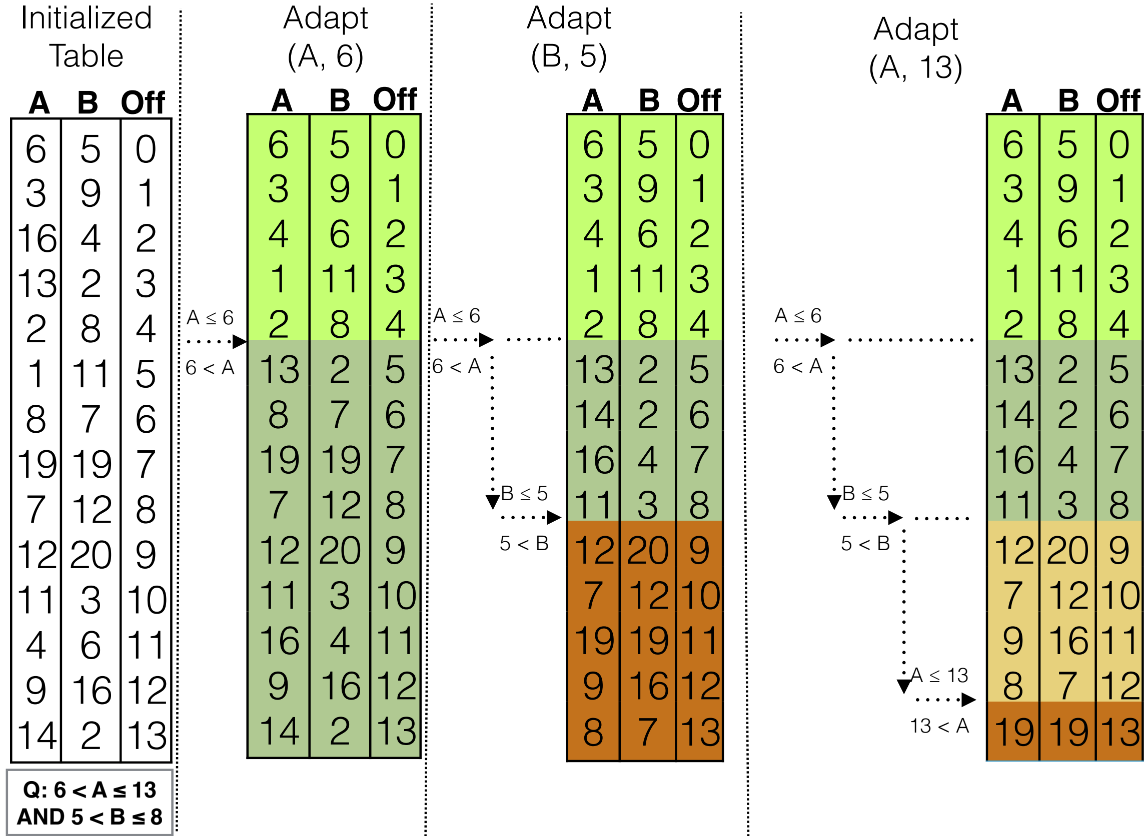


Figure 4-1: Adaptive KD-Tree: The adaptation phase with query: $6 < A \leq 13$ AND $5 < B \leq 8$, and $size_threshold = 4$.

columns is executed. In this phase, it creates a copy of the original table into our index table. In the *adaptation* phase, it swaps rows in the index table to partition it according to the query predicates.

Figure 4-1 depicts an example of the adaptation phase when executing the first query with predicates $6 < A \leq 13$ AND $5 < B \leq 8$ with $size_threshold = 4$. In the first part of our example, we have our initialized index table equal to the original table. In the second step, we start the adaptation phase by generating the attribute-value pairs $(A, 6), (B, 5), (A, 13), (B, 8)$ and partitioning the index table for each of those pairs. In the example, the second step demonstrates the partition of pair $(A, 6)$. We swap the rows of our table, taking 6 as a pivot for the first column A, and insert in the KD-Tree the pivot 6 with the position offset 6. In the third step, we partition the pair $(B, 5)$, where the table is pivoted in the second column B with pivot 5, later on adding it to the KD-Tree. Note that we could perform this partitioning in both the top $(A \leq 6)$ and bottom $(A > 6)$ pieces of our table. However, since the Adaptive KD-Tree only indexes the minimum to answer the query, we only refine the piece

that potentially contains query answers (here, $A > 6$), leaving the piece that surely contains no query answers ($A \leq 6$) unchanged. A similar process is done for the next pair $(A, 13)$ depicted as the fourth step. At this step, the resulting piece size reaches the *size_threshold*, and no further partitioning happens for the last pair $(B, 8)$.

Discussion.

Space-Filling Curve Cracking is the first attempt to perform adaptive indexing of multiple columns. However, as demonstrated by Pavlovic et al. [45], mapping is prohibitively expensive on the first query, excluding this strategy from truly adaptive indexes. QUASII is a more promising solution since it features characteristics that are similar to standard adaptive indexing techniques. However, QUASII's aggressive refinement strategy is undesirable in an adaptive indexing strategy hurting query robustness. Besides, QUASII forces initial queries to pay an unnecessarily high cost. The Adaptive KD-Tree has a less aggressive refinement strategy than QUASII. However, it still does not present the required fined-grained indexing to mitigate the robustness problem, as Progressive Indexing has. Finally, other techniques are self-proclaimed multidimensional adaptive indexes, like AQWA [3] and SICC [59]. However, they do not focus on exploratory data analysis but rather on adaptive indexing for data ingestion. The main goal of AQWA is to adjust for changes in the data in a Hadoop distributed scenario. Simultaneously, SICC mainly focuses on reducing "over-coverage" in entries of frequent data ingestion in streaming systems. Hence, they do not focus on a low penalty for the initial queries, on robustness or index convergence.

3 Multidimensional Progressive Indexing

The *Progressive KD-Tree* is a multidimensional progressive indexing technique inspired by Progressive Quick-Sort (Chapter 3). Like one-dimensional progressive indexing techniques, the main goals of Progressive KD-Tree are to limit the indexing penalty imposed on the first query, achieve robust performance, and ensure deterministic convergence towards a full index — irrespective of the actual query workload or data distribution. We accomplish all three goals by indexing a fixed-size portion of the data with each query, independent of the query predicates. The per-query indexing budget (and hence overhead over a scan) and the convergence speed can be controlled by a parameter δ that determines the fraction of the entire data set indexed with each query. Opting for workload independence, we need to choose the partitioning pivots

independent of the query predicates. We use the average value (arithmetic mean) to yield a reasonably balanced KD-Tree, also with skewed data. Our experiments in Section 4 show that determining the median to guarantee a perfectly balanced KD-Tree is prohibitively expensive and does not pay off. The Progressive KD-Tree follows two phases. In the initial *creation phase*, each query copies a δ fraction of the data out-of-place to our index while pivoting on the first dimension's average value. After all data has been copied, in the subsequent *refinement phase*, queries further split the existing pieces until their size drops below a *size_threshold*. When all pieces reach the qualifying size, we consider that the index has converged to a full index. A fully-converged Progressive KD-Tree will have the same structure as a pre-built full index KD-Tree using arithmetic means as partitioning pivots.

3.1 Data Structure

Listing 3 KD-Node for Progressive Indexing

```
1  template <typename T>
2   struct KDNode {
3      T key;
4      unsigned int discriminator_attribute;
5      struct KDNode* left_child;
6      struct KDNode* right_child;
7      unsigned int start;
8      unsigned int end;
9      unsigned int cur_start;
10     unsigned int cur_end;
11     unsigned int left_sum;
12     unsigned int right_sum;
13  };
```

Instead of using a standard KD-Tree node in our data structure, our Progressive KD-Tree uses a slightly extended KD-Tree node structure, as depicted in Listing 3. The standard elements are a key, a discriminator attribute, and two pointers for the left and right children (Lines 3-6). Since partitioning a single piece (i.e., splitting a single node) can take multiple queries, we cannot simply keep a single offset pointing to the final pivot location. Instead, we need to store the offsets marking the boundaries of the piece at hand (Lines 7-8) as well as the offsets marking the progress of the pivoting so far (Lines 9-10). Once a piece has been fully pivoted, the latter two offsets are identical and mark the pivot's location. In lines 11-12, we define the variables

that sum the value of the next to-be-partitioned dimension. We use these values to calculate the average of each piece for the next dimension, for example, the left pivot is $\frac{left_sum}{cur_start - start}$.

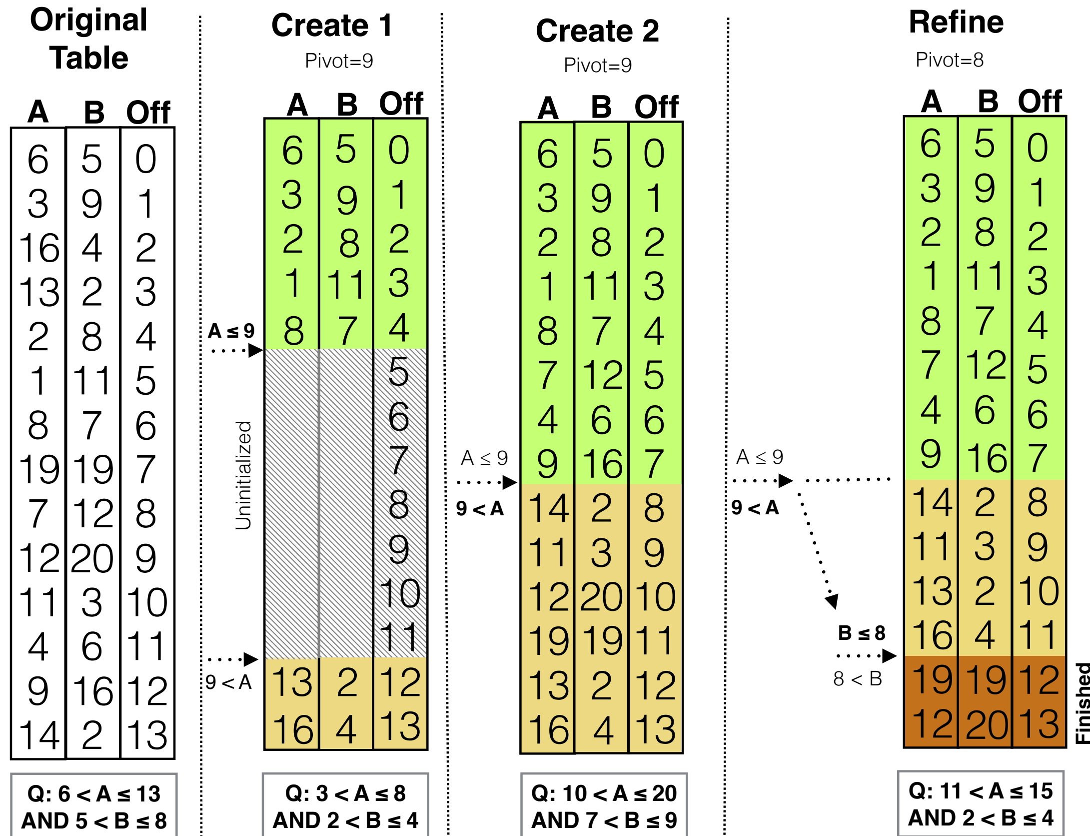


Figure 4-2: Progressive KD-Tree with index budget $\delta = 0.5$ and $size_threshold = 2$. Four queries submitted in the workload.

3.2 Creation Phase

The creation phase copies the data from the original column into our index while filtering and pivoting it on that column's average value. The filtering process is similar to the Adaptive KD-Tree piece scan when copying and pivoting a dimension of the data. We create a candidate list to keep track of elements that qualify its filters. This candidate list is subsequently refined when copying and pivoting the remaining dimensions.

Figure 4-2 depicts an example of the creation phase in the iterations *Create 1* and *Create 2*. In the *Create 1* iteration, we allocate an uninitialized table in DSM format, with columns A and B, having the same size as the original table columns. We

start partitioning in the first dimension A. Unlike the Adaptive KD-Tree, the pivot selection is not impacted by the query predicates. We use the average of that piece's dimension, which is calculated during data loading. In the example, the average value of the whole column A is 9. We then scan the original table and copy the first $N * \delta$ rows to either the top or the bottom of the index, depending on how they compare to the pivot. In our example, we index half of our table, since $\delta = 0.5$. In this step, we also search for any elements that fulfill the query predicate. Afterward, we scan the not-yet-indexed fraction of the original table to answer the query completely. In subsequent iterations, as depicted in *Create 2*, we scan either the top, bottom, or both pieces of the index based on how the query predicate relates to the chosen pivot. In our example, the running query has a filter $3 < A \leq 8$, and we only need to scan the upper piece of our index. Finally, we copy and pivot the other half of our table to our index.

Listing 4 details the creation phase. In lines 2-13, we initialize all necessary variables to compute our candidate list and store elements in the correct position related to the pivot. In lines 2-5, we select the original column, index column, and the query pivots for the dimension discriminated by the root node. In lines 6-7, we create the variables that hold the offsets of both upper and bottom indexed pieces and update the *root.cur_start* and *root.cur_end* after finishing the execution. Line 8 stores an offset to the original table that indicates the last row that was indexed. Line 9 subtracts from our budget the amount of data that will be indexed in this iteration. Lines 11-13 initialize the candidate list that will result from the creation phase and the *go_down* bit vector that for each row keeps track of whether pivoting moves that row to the top part or bottom part of the refined piece. In lines 14-24, the copied elements are indexed, inserting them to either top or bottom of the index while checking if their values match the query predicates (Lines 15 and 16). One might note that all the code is predicated. We avoid branches that could lead to non-robust (i.e., highly varying execution times) due to branch mis-predictions [48, 10]. In line 25, we omit from this listing the code that propagates the pivoting to the remaining dimensions. This code sweeps over each remaining column's respective piece and uses the *go_down* bit vector as set in line 21 to assign each value to the top part or bottom part of the refined piece. The code performs a similar operation to the one described in lines 14-24, with three main differences. First, we do not push elements into the candidate list but rather manage the ones in there while checking for matches in the next dimensions. Second, instead of the pivot comparison, we use the information in the *go_down* bit vector to place the elements in the column properly. Third, for

Listing 4 Code Snippet of the Creation Phase

```
template <class OPL,class OPR> create(Query &q, int& budget) {
    col = orig_tbl.columns[root.dim];
    idx_c = table.columns[root.dim];
    l = q[root.dim].low
    h = q[root.dim].high
    low_pos = root.cur_start;
    high_pos = root.cur_end;
    c_pos = root.cur_start + root.end - root.cur_end;
    n_idx = min(c_pos + budget, root.end);
    budget -= n_idx - c_pos;
    bit_idx = 0;
    CandidateList cl;
    BitVec go_down = BitVec(n_idx-c_pos);
    for (i = c_pos; i < n_idx; i++) {
        mtch = OPL(col[i],l) & OPR(col[i],h);
        cl.maybe_push_back(i,mtch);
        big_pvt = (col[i] >= root.key);
        sml_pvt = 1 - big_pvt;
        idx_c[low_pos] = col[i];
        idx_c[high_pos] = col[i];
        go_down.set(bit_idx++, sml_pvt);
        low_pos += sml_pvt;
        high_pos -= big_pvt;
    }
    ...
    root.cur_start = low_pos;
    root.cur_end = high_pos;
    return cl;
}
```

the first dimension after the *root.dim* we update *root.left_sum* and *root.right_sum* according to the *go_down* bit vector. After indexing all dimensions, in line 26-27, we update the root offsets, and in line 28, we return the created candidate list that refers to the δ fraction of the table. Hence, the function that calls the *create* method still checks the not-yet-indexed fraction of the original table and the previously indexed bottom/top index pieces accordingly.

3.3 Refinement Phase

With the original table no longer required to compute queries, we now perform index lookups. While doing these lookups, we further refine the index pieces until they all have become smaller than a given size threshold, progressively converging towards a full KD-Tree. We focus on refining pieces of the index required for query processing (i.e., pieces containing query pivots). If these pieces are fully refined (i.e., the pieces containing query pivots children reach a size below *size_threshold*) and the indexing budget is not over, refinement is continued on a size priority, refining the largest piece first. The refinement is done by recursively performing quicksort operations to swap rows inside the index. Like the creation phase, we also keep track of the sum left and right children of the indexed piece, which is later used as pivots for the children. After all the refinement for that query is completed, we perform a similar index lookup and piece scan as the Adaptive KD-Tree. The only difference is that we need to also take into account pieces where pivoting is not finished.

Figure 4-2 depicts an example of the refinement phase. In our example, the running query has the filters $10 < A \leq 20$ and $7 < B \leq 9$. A lookup in the index indicates a scan of the bottom piece, and hence that is the piece to be refined on dimension B. We use $\frac{root.right_sum}{root.end - root.current_end}$ value as the pivot. In the example, the pivot is the value 8. With $\delta = 0.5$, we are capable of fully refining that piece around 8. Due to our $size_threshold = 2$, we mark the bottom piece as finished, and no further refinement occurs.

Query Execution

In this section, we describe how we use the Progressive KD-Tree during query execution. In the next paragraphs, we describe the two primary operations, the Index Lookup and the Piece Scan.

Index Lookup. After performing the necessary index creation for the query, we perform an index lookup followed by the scan of all pieces that fit our query predicates. The index lookup starts from the root of the KD-Tree and recursively traverses the tree depending on how the query relates to the current node key. In Figure 4-3 we depict an example of the entire search process for predicates $6 < A \leq 15$ AND $0 < B \leq 5$. The search method starts by comparing the root of the tree that indexes column A on key 6, with the range $6 < A \leq 15$. We need to check the root's right child since both predicate boundaries are greater than the node (i.e., where all elements on $A > 6$ are stored.). We now compare the range $0 < B \leq 5$ to the node that indexes column

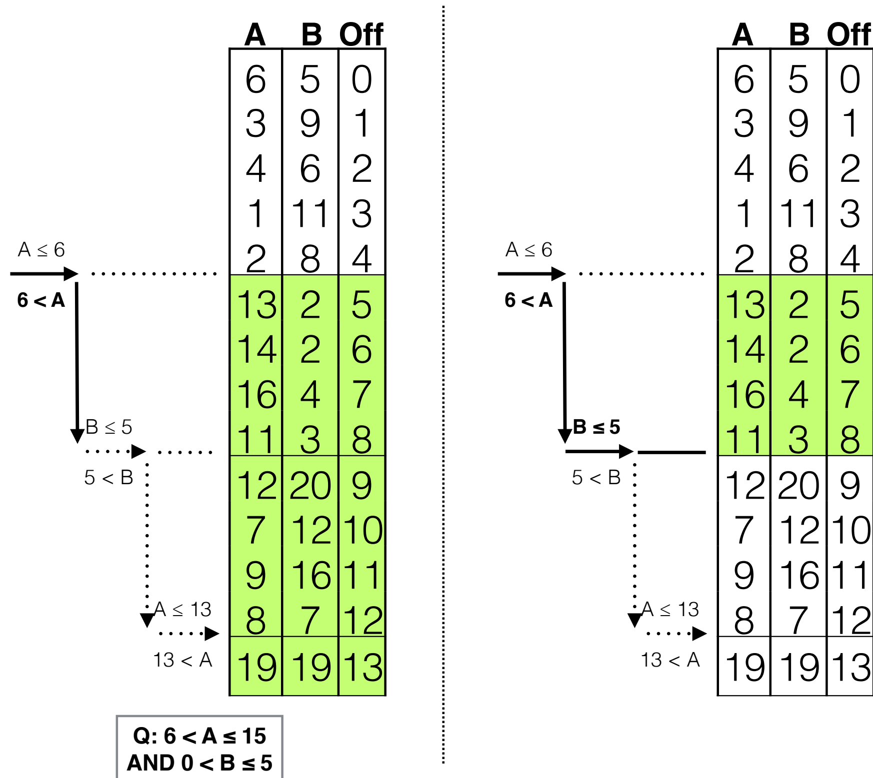


Figure 4-3: A search with predicates $6 < A \leq 15$ AND $0 < B \leq 5$ in the Adaptive KD-Tree.

B on key 5. Note that this time, the predicate boundaries are lower or equal to the node's key. Hence, we only need to check its left child. Finally, since the left child is null, we scan the piece starting on offset 5 until offset 9.

Piece Scan. The index lookup returns a list of pieces that we scan to answer the query. For each piece, we have a pair of offsets indicating where they begin and end and information of which predicates still need to be checked. For example, in Figure 4-3, on the rightmost column, the index would have returned one piece, with offsets 5 and 9. For this piece, we know that all elements in there are $6 < A$ and $B \leq 5$. Hence, we do not need to apply the lower and higher query predicates of attributes A and B, respectively. However, whenever the index does not match our query predicates exactly, we need to perform a multidimensional conjunctive selection on one or more pieces. There are, in general, two ways to perform multidimensional conjunctive selections in column stores. (1) We perform the selection on each column individually, creating an intermediate result per column as (candidate) list of IDs (or as bit-vector). Later, intersecting all lists (or and-ing all bit-vectors) to yield the final result. (2) We

perform the selection over one column, creating an intermediate (candidate) list of IDs (or as bit-vector). Then we use this candidate list (or bit-vector) to test the selection predicate on the next column only for those tuples that qualified with the first column and create a revised candidate list (or bit-vector) as an intermediate result reflecting both selections. We continue accordingly for all remaining columns. Option (1) is advantageous for low selectivities since they focus on sequential scans over the whole data set, while option (2) presents the best performance over high selectivities since, except the first column, we only check elements that qualify. Hence, in all our scans, we use option (2) with a candidate list to achieve the best performance.

Interactivity Threshold.

The user must provide the Progressive KD-Tree with an interactivity time threshold τ and a δ. We distinguish two situations depending on the full scan costs. (1) If a simple scan of the entire table does *not* exceed τ, we use the cost model, presented in the next section, to calculate a δ' such that the first query (incl. indexing a δ' fraction of the data) does not exceed τ. We then use $\delta = min(\delta, \delta')$ for all queries, ensuring that none exceeds τ. (2) If a simple scan of the entire table *does* exceed τ, we use the user-provided δ until the KD-Tree is sufficiently built such that the scan cost per query drops below τ. Then, we calculate a δ' as in situation (1) and proceed as described above.

3.4 Greedy Progressive Indexing

While the δ parameter of Progressive KD-Tree allows us to control both the per-query indexing effort (and hence overhead) and the speed of convergence towards a full index, there is a mutual trade-off. The smaller δ, the lower the overhead, but the slower the convergence; the larger δ, the faster the convergence, but the higher the overhead.

Let t_{scan} denote the time to scan the entire data set, t_{budget} denote the time it takes to pivot/refine a δ fraction of the data set, t'_i denote the net query execution time (i.e., without refining the index) of the ith query Q_i given the current state of the index, and $t_i = t'_i + t_{budget}$ denote the gross execution time (i.e., incl. refining the index) of the ith query Q_i given the current state of the index. The gross execution time t_i of each query with Progressive KD-Tree is bounded by $t_{total} = t_{scan} + t_{budget}$, i.e., $t_i \leq t_{total}$. While this is a tight bound for the first query ($t'_0 = t_{scan} \Rightarrow t_0 = t_{total}$), it gets looser the more queries are being processed and the more of the index is partly constructed, as then the partial index is likely to let queries become faster than a scan

System	ω	cost of sequential page read (s)
	κ	cost of sequential page write (s)
	ϕ	cost of random page access (s)
	σ	cost of random write (s)
	γ	elements per page
Data set & Query	N	number of elements in the data set
	α	% of data scanned in partial index
	d	number of dimensions
Index	δ	% of data to-be-indexed
	ρ	% of data already indexed
	h	height of the KD-Tree

Table 4.1: Parameters for Progressive Indexing Cost Model.

$(t'_i < t_{scan} \Rightarrow t_i < t_{total})$.

While generally decreasing, t'_i, and hence t_i, can still vary significantly until the index is fully built.

We propose *Greedy Progressive KD-Tree* as a refinement of Progressive KD-Tree to ensure that, until the index is fully created, each query Q_i has the same gross execution time $t_i = t_0 = t_{total}$, i.e., exploits the full difference between t_{total} and t'_i for indexing. In this way, we speed-up convergence without increasing total query execution time. To do so, we introduce a *cost model* that estimates the net execution time t'_i for each query Q_i and calculates its maximum indexing budget as $t'_{budget,i} = t_{total} - t'_i$, from which we derive δ'_i for each Q_i. The first query uses the user-provided δ, i.e., $\delta'_0 = \delta \Rightarrow t'_{budget,0} = t_{budget}$.

Cost Model.

The cost model considers the query and the state of the index in a way that is not affected by different data distributions, workload patterns, or query selectivities. In a nutshell, our cost model can tell how much data will be scanned, hence yielding a conservative δ'_i that guarantees that our query cost will never exceed t_{total}. A conservative δ'_i means the highest possible δ'_i in the worst-case, where any construction or refinement does not boost the current query execution. However, if the query execution finishes below the t_{total} limit, we perform one extra step called the *reactive* phase to perform an extra indexing until fully consuming the t_{total} limit. The parameters of the Greedy Progressive KD-Tree cost model are summarized in Table 4.1.

Creation Phase

The total time taken in the creation phase is the sum of (1) the index lookup time (i.e., time to access the root node and decide if we scan the top/bottom of our table), (2) the indexing time, and (3) the original table scan.

(1) To calculate the index lookup time, we need to account for the node access and the top/bottom access of each column of our table, where we perform two random accesses $2 * \phi$,one for the root and one to access the indexed table's first column, and $\frac{\alpha * N}{\gamma}$ for the total data we must scan. Since our data has d dimensions, we must account one random access for the additional columns and multiply the sequential scan by $d-1$. The index lookup time is $t_{lookup} = 2 * \phi + \frac{\alpha * N}{\gamma} + (d-1) * \phi$. Simplifying to $t_{lookup} = \frac{\alpha * N}{\gamma} + (d+1) * \phi$.

(2) The indexing time (i.e., index construction time) consists of scanning the base table pages and writing the pivoted elements to the result array. The indexing time is calculated by multiplying the time it takes to scan and write a page sequentially $(\kappa + \omega)$ by the number of pages we need to write summed with the access of each dimension, resulting in $t_{indexing} = (\kappa + \omega) * \frac{N * \delta}{\gamma} + (d-1) * \phi$.

(3) The original table scan, is given by sequentially reading all not-yet-indexed data. The total fraction of the data that remains unindexed is $1 - \rho - \delta$, hence the scan time of the original table is given by $t_{scan} = \frac{(1-\rho-\delta)}{\gamma} * \omega$.

The total time taken for the creation phase is the sum of all three steps, hence $t_{total} = t_{lookup} + t_{indexing} + t_{scan}$ and we set $\delta = \frac{t_{budget}}{(\kappa+\omega) * \frac{N}{\gamma} + (d-1) * \phi}$.

Refinement Phase

In the refinement phase, we no longer need to scan the base table. Instead, we only need to scan the fraction α of the data in the index. However, we now need to (1) traverse the KD-Tree to figure out the bounds of α, and (2) swap elements in-place inside the index instead of sequentially writing them to refine the index. The height h times the cost of random page access ϕ gives the cost for traversing the KD-Tree, resulting in $t_{lookup} = h * \phi$. For the swapping of elements, we perform predicated (i.e., branch-free) swapping [10] to allow for a constant cost regardless of how many elements we need to swap. The total swap cost is the number of elements we can swap times the cost of swapping them, which is two random writes multiplied by the d dimensions, i.e., $t_{swap} = N * \delta * 2 * d * \sigma$. The total cost in this phase is therefore equivalent to $t_{total} = t_{lookup} + \alpha * t_{scan} + t_{swap}$. Finally, we set $\delta = \frac{t_{budget}}{N * 2 * d * \sigma}$ for the adaptive indexing budget.

Interactivity Threshold

With Greedy Progressive KD-Tree, in addition to the mandatory interactivity time threshold τ, the user can additionally provide a "penalty" budget δ or a limit x of queries. We distinguish two situations, depending on the full scan cost. (1) If $t_{scan} < \tau$, we set $t_{total} = \tau$, i.e., ensure that no query exceeds τ, and use our cost model to calculate all $t_{budget,i}$ and δ'_i (incl. the first query's $t_{budget,0}$ and δ'_0) as described above. In this case, we ignore the also provided δ or x. (2) If $t_{scan} \geq \tau$, we distinguish two cases. (2a) In case the user provided a "penalty" budget δ, we start with $t_{total} = t_{scan} + t_{budget}$ with δ, and use our cost model to calculate all $t_{budget,i}$ and δ'_i until the KD-Tree is sufficiently built such that the scan cost per query drop below τ.

(2b) In case the user provided a limit x of queries, we use our cost model to calculate the amount of indexing that is required to build a partial KD-Tree such that the remaining scan cost per query is less than τ, distribute this indexing work over x queries, and calculate how much indexing budget $t_{budget++}$ is needed for each query. With this, we proceed as in (2a) for the first x queries. In both cases, (2a) & (2b), we then proceed with the user-provided τ as in situation (1).

4 Experimental Analysis

In this section, we provide a quantitative assessment of our proposed progressive indexes. This section is divided into four parts. First, we define all real and synthetic data sets and workloads used in our assessment. Second, we analyze the impact of δ on the Progressive KD-Tree in terms of first query cost, pay-off, time until full convergence, and total time. Third, we provide an in-depth performance comparison of our proposed progressive indexes and analyze their behavior under three real and eight synthetic workloads. We also provide comparisons with the state-of-the-art on multidimensional adaptive indexes QUASII (Q) and Adaptive KD-Tree (AKD). We use two variations of a full KD-Tree index as a baseline. The first one using the average value of a piece as the pivot (AvgKD), and the second one using medians (MedKD). Finally, we study our algorithms' behavior when the full scan cost is higher than the interactivity threshold.

4.1 Setup.

All indexes were implemented in a stand-alone C++ program. All the data is 4-byte floating-point numbers stored in a columnar format (i.e., DSM). The code was

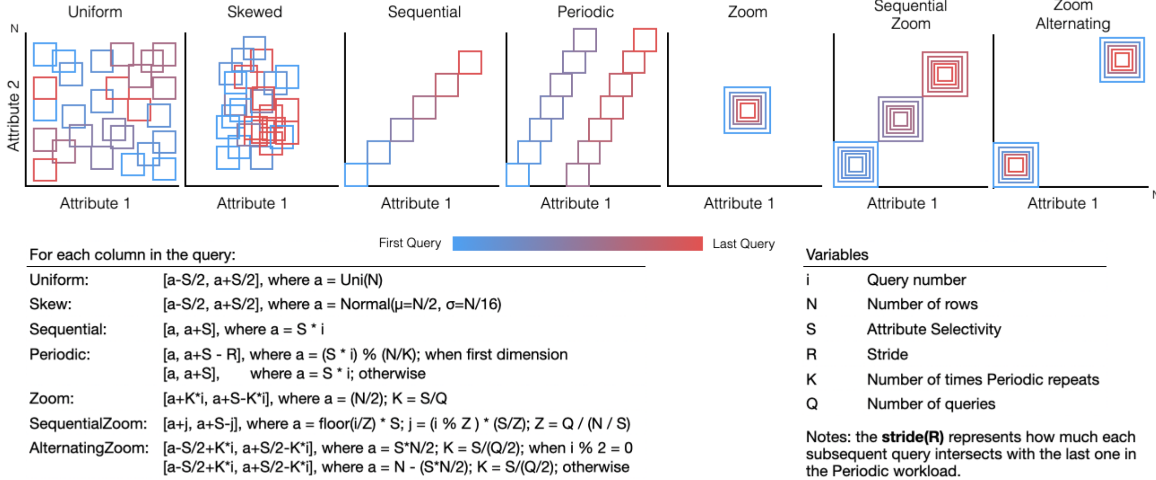


Figure 4-4: Visual representation of the different synthetic workloads.

compiled using GNU g++ version 9.2.1 with optimization level -O3. All experiments were conducted on a machine with 256 GB of main memory, an Intel Xeon E5-2650 with 2.0 GHz clock, and 20 MB of L3 cache size.

4.2 Data Sets & Workloads

We use four different data sets in our assessment.

Power. The power benchmark consists of sensor data collected from a manufacturing installation, obtained from the DEBS 2012 challenge[3]. The data set has three dimensions and 10 million tuples. The workload consists of random close-range queries on each dimension, a total of 3000 queries.

Skyserver. The Sloan Digital Sky Survey is a project to map the universe. Their data and queries are publicly available at their website[4]. The data set we use here consists of two columns, *ra* and *dec*, from the *photoobjall* table with approximately 69 million tuples. The workload consists of 100,000 real range queries executed on those two attributes.

Genomics. The 1000 Genomes Project collects data regarding human genomes. It consists of 10 million genomes, described in 19 dimensions. The workload consists of 100 queries constructed by bio-informaticians.

Uniform. It follows a uniform data distribution for each attribute in the table, consisting of 4-byte floating-point numbers in the range of $[0, N)$, where N is the experiment's number of tuples. We use eight different synthetic workloads in our performance comparison, similar to those described in Chapter 3 but extended for

[3]https://debs.org/grand-challenges/2012/
[4]http://skyserver.sdss.org

the multidimensional case. Figure 4-4 depicts a two-dimensional example of these workloads with the mathematical formulas used to generate them. In addition to these workloads, we propose a new one, called *shifting*. The shifting workload represents a common scenario in data exploration where the columns being queried change constantly (e.g., the data scientist executes ten queries on three columns, which leads him to investigate the other three columns, and so forth). When generating a synthetic workload, we take as a parameter the overall query selectivity σ. To keep σ constant, regardless of the number d of dimensions used, we set the per-dimension selectivity with d dimensions to $\sigma_d = \sqrt[d]{\sigma}$; e.g., for $\sigma = 1\%$, we get $\sigma_2 = 10\%, \sigma_4 = 31\%, \sigma_6 = 46\%, \sigma_8 = 56\%$.

4.3 Delta Impact

The parameter δ defines a percentage of the total amount of our data that is pivoted per query. If $\delta = 0$, no indexing is performed, hence only full scans are executed, and the index will never converge. On the other hand, if $\delta = 1$, the creation phase completes in the first query, with the data fully pivoted once in the first dimension. In this section, we explore how δ impacts our index in terms of the burden on the first query, how many queries it takes for the index to pay-off when compared to a full scan, how much time it takes until full index convergence, and the impacts on cumulative time for the entire workload. We use a uniform data set and workload, with 30 million rows, $d \in \{2, 4, 6, 8\}$ columns, and 1000 queries with 1% selectivity. We test with multiple δ values, ranging from 0.1 to 1. Where applicable, we compare Progressive KD-Tree (PKD) with Adaptive KD-Tree (AKD), QUASII (Q), Average/Median KD-Tree (AvgKD/MedKD), Full Scan (FS). Both Average and Median KD-Tree are built using the attribute order given by the table schema.

First Query

The first query cost is the cost of fully scanning the data with the addition of copying and pivoting a δ-fraction of the data. Figure 4-5 depicts the first query cost over varying δ for multiple columns. With Progressive KD-Tree, the cost increases linearly as we increase δ, and hence the amount of indexed data, with the impact being larger, the more columns are involved, i.e., the more data needs to be copied. With $\delta = 0$, the first query merely performs a Full Scan. The first query cost for Adaptive KD-Tree is about the same as for Progressive KD-Tree with $\delta \in [0.6, 0.7]$. The first query cost of QUASII is significantly higher than those of both Adaptive and Progressive KD-Tree

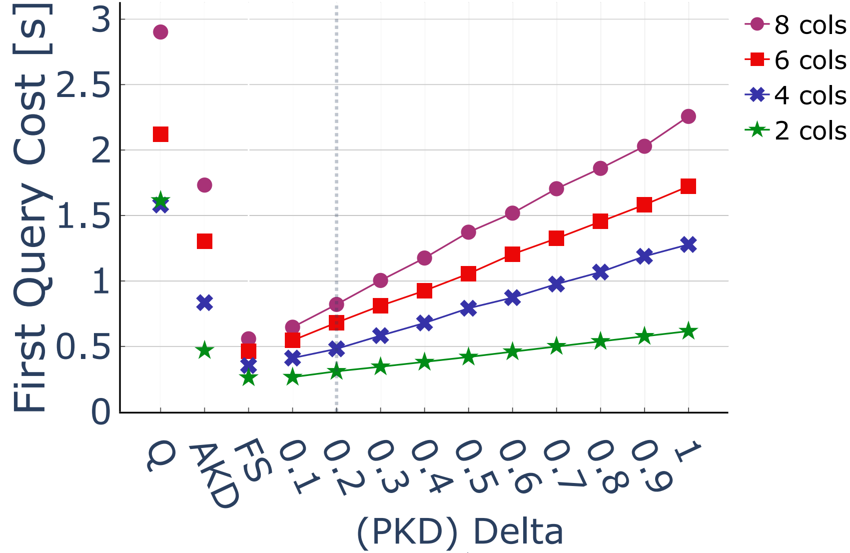


Figure 4-5: First query cost.

due to the more intensive refinement work of QUASII. For Average KD-Tree and Median KD-Tree, the first query costs grow linearly with the number of columns. We omit them from Figure 4-5 as building the entire index is far more expensive than any query shown there.

Pay-Off

In this experiment, we define pay-off as the number q of queries required until investing in incrementally building the Progressive KD-Tree pays off compared to performing only full scans without indexing, i.e., the smallest q for which $\sum_{i=0}^{q} t_{i,progKD} \leq \sum_{i=0}^{q} t_{i,FScan}$. Figure 4-6 depicts the pay-off for multiple dimensions. While a small δ limits the indexing impact over the initial queries, it also limits and the indexing progress. For workloads with high per-column selectivity, this results in the queries being capable of taking advantage of the little index progress early on. However, in a workload with a low per-column selectivity (e.g., with 8 columns, we need a per-column selectivity of 56% to yield an overall query selectivity of 1%), this results in the queries not being able to take advantage of the indexing early on. For example, with $\delta = 0.1$, it takes 10 queries to pivot the first node fully. Since in our experiment, we use a uniform data set, and the Progressive KD-Tree uses averages as pivots, that results in a pivot that partitions the data on two pieces with approximately 50% of the total data. In the case of an 8-dimensional workload with per-column selectivity of 56%, the

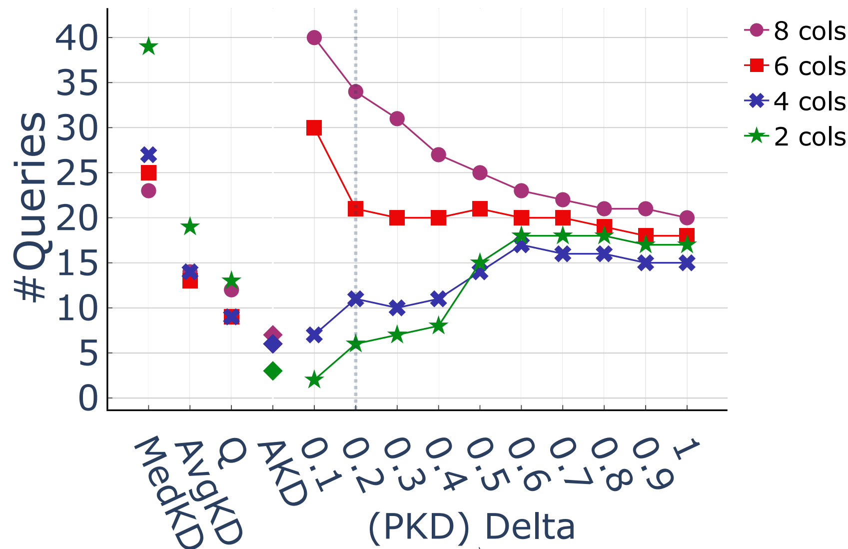


Figure 4-6: #Queries until Pay-off.

workload cannot take advantage of the index for the first 10 queries. Hence, the initial queries always perform index creation and full scans, resulting in a higher pay-off when compared to lower per-column selectivities. Furthermore, a higher δ reduces the limitation on the index progress, creating an index that can boost queries early on and diminishing the number of queries for the pay-off. Regarding the minimal indexing for the given workload, Adaptive KD-Tree pays-off as early as the quickest variant of Progressive KD-Tree ($\delta = 0.1$).

Convergence

The convergence is defined as the time, in seconds, it takes for the Progressive KD-Tree to fully index the data and achieve the same query performance as the Average KD-Tree. Figure 4-7 depicts the convergence for multiple dimensions. The time to converge increases with the number of dimensions because the average query time also increases. However, since δ determines a percentage of the data that is indexed per query, the number of dimensions has no impact on the number of queries to converge. For example, with $\delta = 0.1$, the number of queries to converge is about 100, independent of the number of columns.

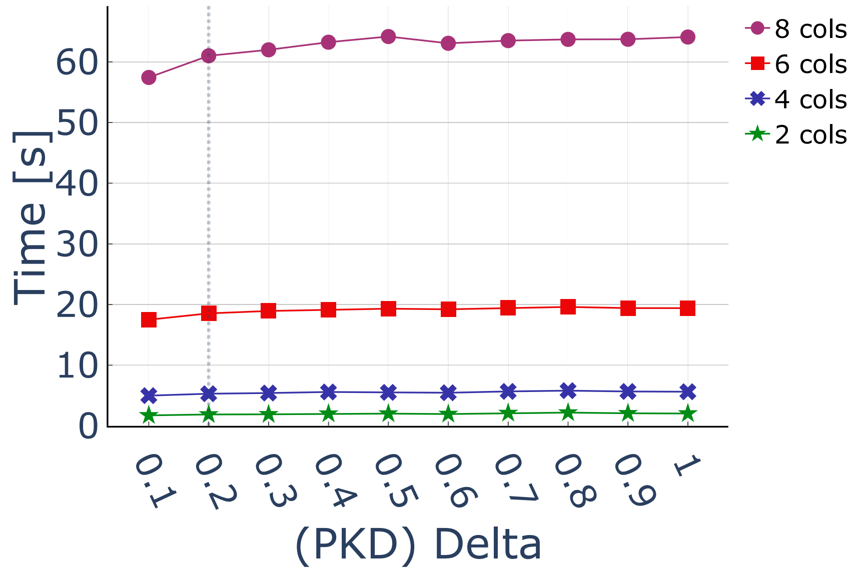


Figure 4-7: Time until Convergence.

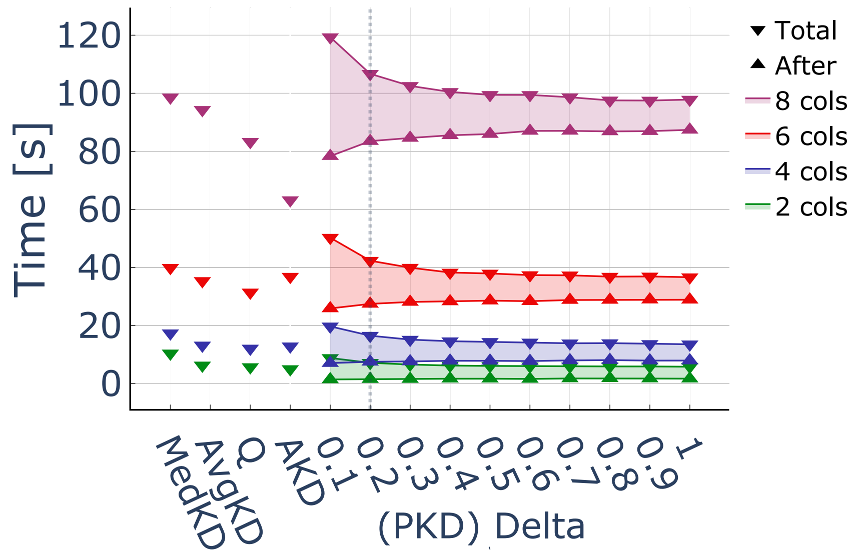


Figure 4-8: Cumulative time (1000 queries).

Cumulative Time

In Figure 4-8, downward-pointing triangles ("Total") mark the cumulative times to execute the entire workload of 1000 queries, while upward-pointing triangles ("After") mark the cumulative times for only the tail of the workload after the index is fully built and used for optimal query performance, i.e., no further index refinement is performed.

The shaded range between both indicates the cumulative time until the index is fully built. Progressive KD-Tree takes at most 103 queries to converge to a full index with $\delta = 0.1$, or even as a mere 10 queries with $\delta = 1$. Consequently, 90% ($\delta = 0.1$) to 99% ($\delta = 1$) of the 1000 queries in the workload benefit from the fully-built index, accounting for the majority of the cumulative execution time due to their number rather than per-query time. Only between 1% ($\delta = 1$) and 10% ($\delta = 0.1$) of the workload contribute to progressively constructing the index. For the non-progressive techniques, we only show the "Total" workload time without breaking it down into before and after convergence. Adaptive KD-Tree and QUASII never converge in this experiment, while Average KD-Tree and Median KD-Tree converge with the first query by design. Overall, with $\delta \geq 0.2$, Progressive KD-Tree yields about the same total workload time as the non-progressive techniques. Only in the 8-dimensional scenario, QUASII and Adaptive KD-Tree outperform Progressive KD-Tree.

Picking a Delta (δ).

For exploratory data analysis, our indexes must not impose a high burden over the initial queries while still paying off their investments quickly and preferably converging fast and presenting a low total cost. Taking these objectives in mind, we select a $\delta = 0.2$ for our performance comparisons. It offers a sharp decrease in total cost and convergence compared to $\delta = 0.1$, without a significant increase in cost in the first query.

4.4 Performance Comparison

In the remainder of the experimental section, we will focus on comparing the performance of the Progressive KD-Tree (PKD) and the Greedy Progressive KD-Tree (GPKD) with the state-of-the-art. In particular, we compare it with QUASII (Q), Adaptive KD-Tree (AKD), and two KD-Tree full-index implementations, the Average KD-Tree (AvgKD) that uses the average value of pieces as pivots and the median KD-Tree (MedKD) that uses the median values as pivots. We also test a Full Scan (FS) implementation using candidate lists as the baseline.

We verify four main characteristics that are desirable in indexing approaches for multidimensional exploratory data analysis. (1) The first query cost. (2) The number of queries executed, so the investment performed on index creation pays-off. (3) The workload robustness. (4) The total workload cost. To evaluate our indexes, we execute all workloads as described in Section 4.2.

We execute the real workloads as given. For the Synthetic workloads, we generate $d = 8$ dimensions, with 300 million tuples for Uniform, Skewed, SequentialZoom, and 50 million tuples for all others. All queries have $\sigma = 1\%$ overall selectivity, while the per-dimension selectivity for all columns is $\sigma_8 = 56\%$. The only exception is the sequential workload, where we only generate two dimensions with $\sigma_2 = 0.1\%$. This is because, with the sequential workload, query ranges must not overlap; with more than two attributes, the per attribute selectivity is too big, and using query selectivity $\sigma = 1\%$ would yield only 10 disjoint queries. Hence, we decrease overall selectivity to $\sigma = 0.0001\%$, which yields 1000 disjoint queries.

We use *size_threshold* = 1024 tuples as a minimum partition size for all indexes. Unless stated otherwise, all progressive indexing experiments use an interactivity threshold equal to the first query cost of PKD with $\delta = 0.2$.

First Query.

		MedKD	AvgKD	Q	AKD	PKD(.2)	GPKD(.2)	FS
50M	Unif(8)	20.20	12.46	5.11	3.07	1.36	1.36	0.91
	Skewed(8)	20.23	12.48	6.25	3.49	1.26	1.26	0.82
	Zoom(8)	20.28	12.68	6.13	3.24	1.32	1.31	0.84
	Prdc(8)	20.17	12.42	6.99	6.94	0.99	1.00	0.60
	SeqZoom(8)	19.98	12.42	5.23	2.90	1.42	1.41	0.93
	AltZoom(8)	20.18	12.43	6.98	6.93	0.99	1.00	0.60
	Shift(8)	20.20	12.46	5.11	3.07	1.36	1.36	0.91
	Seq (2)	15.88	8.30	4.01	0.68	0.26	0.26	0.19
Real	Power	1.52	0.83	0.33	0.23	0.08	0.08	0.06
	Genomics	2.58	2.62	1.25	0.99	0.27	0.27	0.03
	Skyserver	14.31	6.84	1.19	0.63	0.36	0.35	0.26
300M	Unif(8)	146.72	83.91	37.25	20.93	8.17	8.17	5.47
	Skewed(8)	146.80	84.01	43.06	21.24	7.94	7.96	5.12
	SeqZoom(8)	146.87	84.36	35.93	18.08	8.84	8.83	6.41

Table 4.2: First query response time (Seconds).

Table 4.2 depicts the first query cost of all algorithms on all workloads. The Median KD-Tree and the Average KD-Tree present the highest times on the first query since they create a full index when we query a group of columns for the first time. The Median KD-Tree usually presents a higher cost since finding the median of a piece is more costly than finding the average value. The adaptive indexes are up to one order of magnitude cheaper than the full indexes since they only index a focused region necessary to answer the query. QUASII has a more aggressive partitioning

algorithm than the Adaptive KD-Tree (for example, in the first query of the uniform workload, the Adaptive KD-Tree creates 161 nodes while QUASII creates 13,480) and, thus, ends up being a factor 2 slower in the first query evaluation. Finally, both progressive indexing solutions have the same time on the first query since they execute it with the same δ. They impose the smallest burden on the first query and are up to one order of magnitude faster than the adaptive indexing solutions.

Pay-off.

		MedKD	AvgKD	Q	AKD	PKD(.2)	GPKD(.2)
50M	Unif(8)	22.19	13.57	11.12	6.83	31.41	22.88
	Skewed(8)	23.67	14.42	9.90	5.44	36.06	28.06
	Zoom(8)	31.25	18.54	6.19	3.26	39.50	30.19
	Prdc(8)	22.00	13.47	7.08	7.09	29.14	22.53
	SeqZoom(8)	21.22	13.20	5.27	2.91	32.00	24.39
	AltZoom(8)	21.53	13.15	8.12	7.57	19.15	26.46
	Shift(8)	2094.98	1319.28	1085.27	26.34	1152.43	1263.61
	Seq (2)	15.89	8.30	4.07	51.17	1.93	7.62
Real	Power	1.79	0.96	0.81	0.41	1.04	1.80
	Genomics	6.41	6.49	9.06	6.09	16.16	17.69
	Skyserver	14.32	6.84	1.24	0.75	2.91	9.40
300M	Unif(8)	154.82	87.70	74.92	40.52	197.89	160.04
	Skewed(8)	159.33	88.26	65.96	32.97	229.73	180.63
	SeqZoom(8)	151.92	91.32	36.17	18.17	185.14	155.27

Table 4.3: Pay-off (Seconds).

Table 4.3 depicts the time it takes for the investment spent on index creation to pay-off when compared to a full scan. For the full index approaches, the Average KD-Tree presents a smaller pay-off than the Median KD-Tree due to a lower cost on index creation while maintaining a similar cost on index lookup. In the adaptive solutions, the Adaptive KD-Tree has the lowest pay-off, not only when compared to QUASII, but overall, this is a direct result of its core design of only indexing the pieces necessary for the executing query. At the same time, QUASII performs a more aggressive refinement strategy that increases its pay-off. The Adaptive KD-Tree has the worst pay-off in the sequential workload, which represents its worst-case scenario. Finally, the progressive solutions present the highest pay-off in general. However, it is important to notice that we picked our δs optimizing for a low burden in the first query. Since most experiments are with 8 columns, as depicted in Figure 4-6 to optimize for a low pay-off we would need to use larger δs. One can notice that

the progressive solutions perform the best on the sequential workload due to the low number of columns benefiting from the small δ. One can notice that for the Shift(8) workload, no algorithm besides the Adaptive KD-Tree pays-off due to the low number of queries executed before shifting the columns we are looking into. Figure 4-9 depicts

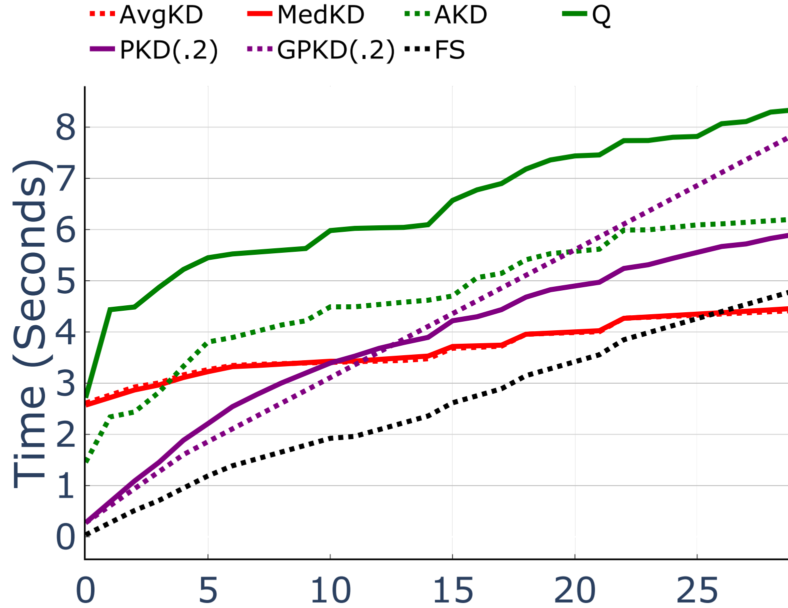


Figure 4-9: Cumulative response time.
Genomics, first 30 queries.

the cumulative response time of the first 30 queries in the Genomics Benchmark. Compared to full indexes, both adaptive and progressive indexes take longer to pay-off and achieve full index response time. This is due to the full indexes having a low first query cost, as discussed in the first query sub-section.

Robustness.

To calculate the robustness, we check the variance in per-query cost, for the first 50 queries or up to full index convergence. For full indexes, the variance is 0 because it fully converges in the first query. Table 4.4 depicts the robustness of all adaptive and progressive algorithms. The Adaptive KD-Tree is as robust as QUASII. The progressive indexing solutions are the most robust options, with up to 3 orders of magnitude lower variance than the adaptive indexing approaches, with the Greedy Progressive KD-Tree always being the most robust, with a constant per-query cost until convergence due to its cost model adaptive δ (Fig. 4-10).

		Q	AKD	PKD(.2)	GPKD(.2)
50M	Unif(8)	6E-01	2E-01	9E-02	1E-03
	Skewed(8)	8E-01	2E-01	8E-02	2E-03
	Zoom(8)	7E-01	2E-01	8E-02	1E-03
	Prdc(8)	1E+00	9E-01	4E-02	6E-04
	SeqZoom(8)	5E-01	2E-01	1E-01	2E-03
	AltZoom(8)	1E+00	9E-01	8E-02	6E-04
	Shift(8)	2E+00	9E-01	3E-02	1E-03
	Seq (2)	3E-01	3E-03	1E-03	8E-05
Real	Power	3E-03	1E-03	6E-04	3E-05
	Genomics	2E-01	6E-02	1E-02	9E-04
	Skyserver	4E-02	8E-03	4E-03	2E-04
300M	Unif(8)	3E+01	1E+01	4E+00	3E-02
	Skewed(8)	4E+01	9E+00	3E+00	3E-02
	SeqZoom(8)	3E+01	6E+00	4E+00	5E-02

Table 4.4: Query time variance (smaller is better).

Figure 4-10: Per query response time.
Uniform(8), first 50 queries.

Total Response Time.

Table 4.5 depicts the total response time of all benchmarks. The Progressive Indexing approaches have a very similar response time compared to the full indexes due to their design characteristics prioritizing robustness and convergence over total response time, which is reinforced by the low δ picked for the experiments. Adaptive indexing always

		MedKD	AvgKD	Q	AKD	PKD(.2)	GPKD(.2)	FS
50M	Unif(8)	109.7	101.4	95.6	74.3	122.6	109.9	857.5
	Skewed(8)	147.6	138.3	107.6	43.1	160.8	151.1	856.6
	Zoom(8)	52.0	40.9	11.4	7.1	58.5	51.6	687.1
	Prdc(8)	85.8	73.6	61.9	229.9	93.3	86.4	807.7
	SeqZoom(8)	31.0	24.2	8.2	4.5	46.6	34.1	499.6
	AltZoom(8)	44.0	34.2	18.9	22.4	53.4	48.3	747.0
	Shift(8)	2095.0	1319.3	1085.3	775.5	1152.4	1263.6	885.5
	Seq (2)	15.9	8.3	6.0	102.9	7.8	7.6	332.6
Real	Power	26.0	24.4	24.6	31.3	25.0	24.7	164.6
	Genomics	10.9	10.9	10.6	7.3	16.2	17.7	16.1
	Skyserver	16.0	14.1	6.9	12.0	10.7	10.4	20186.5
300M	Unif(8)	468.8	366.9	422.9	352.0	558.4	472.7	5423.8
	Skewed(8)	581.9	399.8	521.0	195.2	674.9	595.9	5367.1
	SeqZoom(8)	183.0	122.5	48.7	24.5	277.3	186.0	3221.2

Table 4.5: Total response time (Seconds).

has the lowest total response time due to its high focus on refining pieces requested by the currently executing query. The Adaptive KD-Tree presents the fastest results for most of the workloads. The exception is for highly skewed workloads (e.g., Alternating Zoom and SkyServer), which is due to QUASII's extra refinement paying-off almost immediately, and in the Periodic and Sequential Benchmarks.

The Sequential benchmark emulates the worst-case scenario for the Adaptive KD-Tree, where the KD-Tree ends up almost equal to a linked list. This happens due to blindly adapting using the query predicates and because the KD-Tree has no self-balancing mechanism.

The Shifting benchmark also presents a peculiar result. The only index with a faster response time than the full scan is the Adaptive KD-Tree, with its workload-dependent refinement approach quickly paying off for such a small window of queries.

4.5 Impact of Dimensionality

In this section, we evaluate how the number of dimensions affects the performance of each technique. We experiment with a uniform workload of 1000 queries with 1% selectivity on a uniform data set with 2, 4, 8, and 16 columns. Table 4.6 depicts the first query cost, time to pay-off, time until convergence, robustness, and total execution time for each index. Similar to the results presented in the previous section, the Average KD-Tree has the upper hand in terms of total cost and number of queries until pay-off, while the Progressive KD-Trees are the most robust with a predictable

		MedKD	AvgKD	Q	AKD	PKD(.2)	GPKD(.2)	FS
Unif(2)	First Query	15.94	8.35	2.89	1.05	0.55	0.54	0.52
	PayOff	16.05	8.40	5.56	1.63	1.94	8.18	-
	Convergence	-	-	*	*	9.68	7.78	-
	Robustness	-	-	0.20	0.02	0.01	0.00	-
	Time	19.08	11.49	10.76	9.34	12.75	11.24	425.34
Unif(4)	First Query	17.13	9.56	3.14	1.65	0.83	0.82	0.65
	PayOff	17.33	9.66	5.80	3.26	4.65	11.40	-
	Convergence	-	-	*	*	14.47	10.66	-
	Robustness	-	-	0.20	0.08	0.03	0.00	-
	Time	25.27	17.72	17.13	18.32	22.32	19.39	614.59
Unif(8)	First Query	20.20	12.46	5.11	3.07	1.36	1.36	0.91
	PayOff	22.19	13.57	11.12	6.83	31.41	22.88	-
	Convergence	-	-	*	*	38.02	21.34	-
	Robustness	-	-	0.60	0.20	0.09	0.00	-
	Time	109.69	101.41	95.59	74.27	122.60	109.90	857.54
Unif(16)	First Query	45.10	36.99	29.19	10.85	2.07	2.05	1.30
	PayOff	223.96	173.06	50.65	35.64	183.21	185.68	-
	Convergence	-	-	*	*	96.14	74.17	-
	Robustness	-	-	20.00	3.00	0.03	0.08	-
	Time	1054.69	1023.24	461.45	260.02	1026.44	1029.89	1258.90

Table 4.6: Performance difference on Uniform benchmark with different number of attributes.

convergence. One can notice that as the number of dimensions increases, the difference in total time and pay-off between the Adaptive Indexing solutions and the Progressive Indexing increases drastically. This happens due to the convergence principle of progressive indexing, which causes it to behave similarly to a full index.

4.6 Full Scan Exceeding the Interactivity Threshold

Figure 4-11 depicts the behavior of the Adaptive KD-Tree (AKD), the Progressive KD-Tree (PKD), and both options for the Greedy Progressive KD-Tree, with a fixed number of queries as input (GPFQ) and a fixed penalty (GPFP). For this experiment, we set our interactive threshold to 0.5s, approximately half the cost of a full scan. AKD performs the necessary indexing as a pre-processing step during the first query. Hence its first query is one order of magnitude more expensive than a full scan. Due to this investment, all remaining queries are under the threshold. PKD starts with the user-provided δ of 0.2 and gradually reaches a scan cost below the interactivity threshold. At that point, it calculates a new δ', which gradually converges to a full

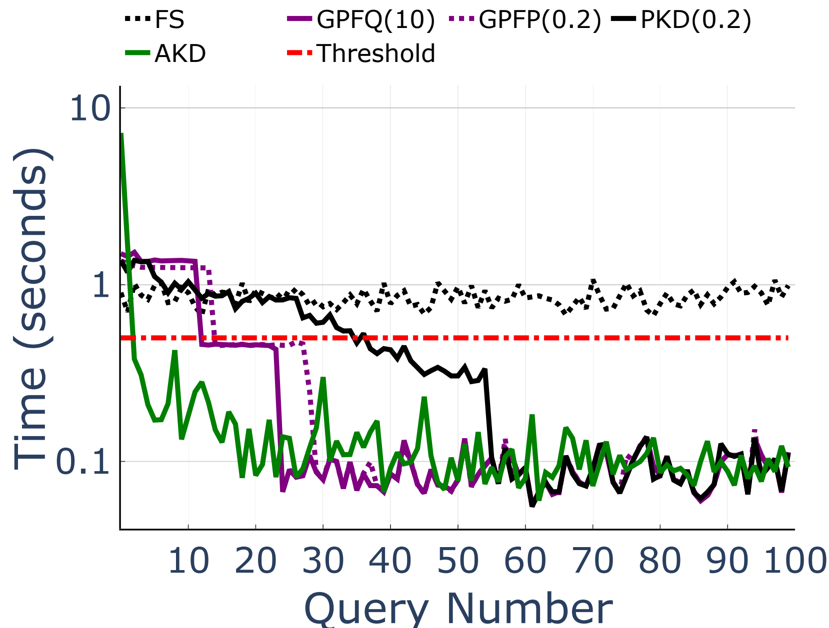


Figure 4-11: Adaptive and Progressive KD-Tree with scans costs exceeding the interactivity threshold; first 100 queries.

index. Both GPFQ and GPFP have similar behavior. They start at a cost higher than the interactivity threshold, have a sudden drop to the threshold cost, and later one more drop until full convergence. For GPFQ, this first drop happens after ten queries, as requested by the user, at the expense of slightly higher first query costs than GPFP. GPFP uses an indexing penalty of $\delta = 0.2$, and only drops once pieces are small enough, slightly later than GPFQ.

5 Summary

This chapter extended existing work on multidimensional adaptive indexing by introducing two new progressive indexing algorithms. We showed that our algorithms are superior compared with state-of-the-art multidimensional indexing in various real and synthetic workloads. In summary, both Progressive KD-Tree's present the lowest penalty on the initial queries, with the Greedy Progressive KD-Tree yielding the fastest convergence and best robustness. In general, which technique to use depends on the properties desired by the user. If the ultimate goal is the total cost, the Adaptive KD-Tree is the algorithm of choice. However, in exploratory data analysis, where we want to keep the impact on initial queries low, and we want a constant query response time without performance spikes, Greedy Progressive KD-Tree is the logical choice.

Up to this point in this thesis, we explored how to create uni and multidimensional

Progressive Indexes. However, these indexes assume that the data is immutable (i.e., no appends or updates happen). In the next chapter, we propose one new progressive algorithm designed to merge updates into Progressive Indexes.

CHAPTER 5

Progressive Merges

1 Introduction

The major drawback of Progressive Indexes is that they are only designed for static databases. However, in the interactive data analysis scenario, the data is not static but rather frequently updated with batches of data that must be appended. If we take the flight dataset example presented in Chapter 2 we can consider the scenario where batches of data are regularly appended since new flights happen all the time (e.g., either data is appended every few minutes, hours, days, depending on how critical is to analyze recent data).

One way of adapting the current Progressive Indexing strategy to support updates is to use the techniques developed for merging updates on Adaptive Indexes since they produce similar intermediate incremental indexes. However, these merging techniques follow Adaptive Indexing's philosophy of lazy query execution, drastically decreasing robustness (i.e., it creates performance spikes that vary the per-query response time in orders of magnitudes up and down), with no guaranteed convergence and high penalties for larger batches of appends.

In this chapter, we introduce *Progressive Mergesort*. Progressive Mergesort is designed to efficiently merge batches of appends while following Progressive Indexing's core design decisions. It presents a low-query impact even for large batches, high robustness, and guaranteed convergence (i.e., all elements are merged into one array).

1.1 Contributions

The main contributions of this chapter are:

- We introduce a novel Progressive Indexing technique that focuses on merging batches of appends into our main Progressive Indexing run.
- We experimentally verify that the Progressive Mergesort provides a more robust, predictable, and faster performance through various batch sizes and update frequencies.
- We provide Open-Source implementations of Progressive Mergesort.[1]

1.2 Outline

This chapter is organized as follows. In Section 2, we investigate related research on updating Adaptive Indexes, called Adaptive Merges. In Section 3, we describe our novel Progressive Mergesort technique and discuss its benefits and drawbacks. In Section 4, we perform an experimental evaluation of each of the novel methods we introduce, and we compare it against adaptive merging techniques. Finally, in Section 5 we draw our conclusions.

2 Related Work

There are three main algorithms designed to efficiently merge appends into adaptive indexes [34], the *Merge Complete*, *Merge Gradual*, and *Merge Ripple*, and we will refer to these algorithms as *Adaptive Merges* from now on. They follow the same philosophy of Adaptive Indexing by only merging appends when necessary. They differ from each other in terms of what data they will merge and how they merge it. In the following subsections, we overview each algorithm and present an example of their execution. Besides the strategies to efficiently merge appends into the index's column, Holanda et al. [29] presents a strategy to prune cold data from the cracker index to boost updates. However, we do not explore this strategy in this work since it directly goes against our full convergence philosophy.

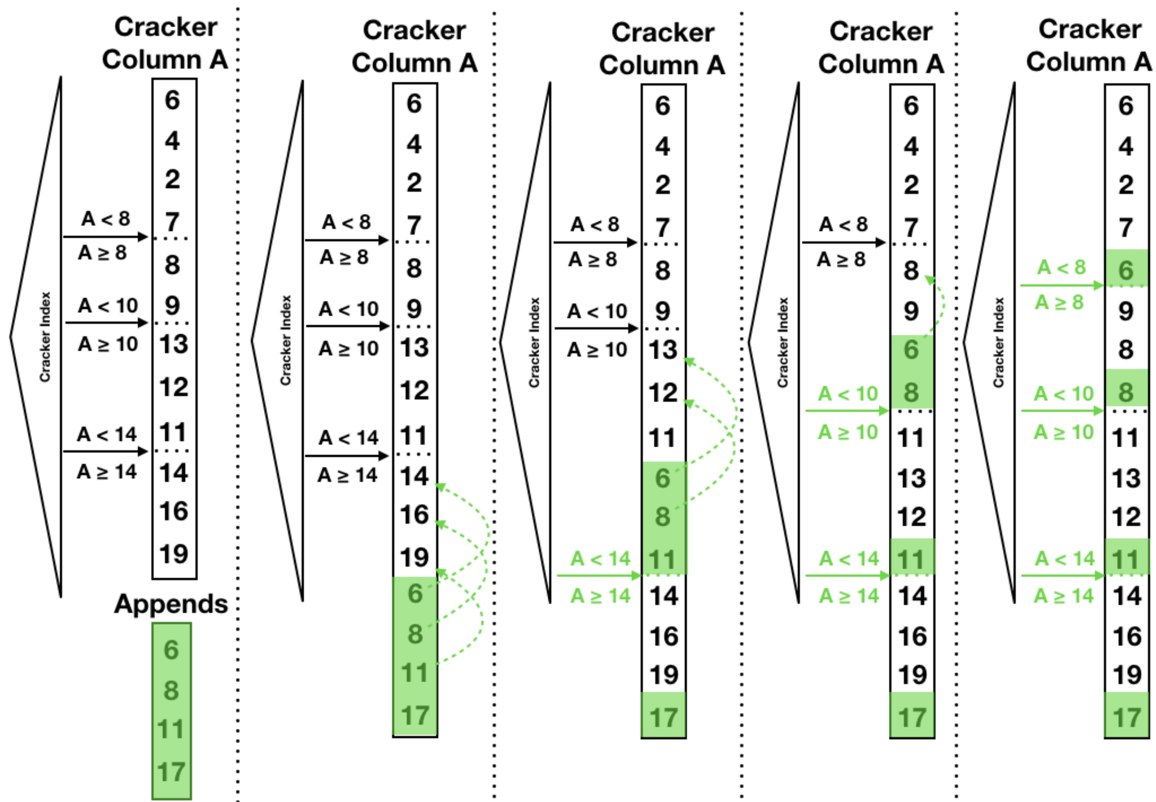


Figure 5-1: Merge Complete on query A ¡ 8

2.1 Merge Complete (MC)

This algorithm completely merges the full *Appends* vector into the *Cracker Column* as soon as a query requests data that is also present in the *Appends* vector.

Figure 5-1 depicts an example of merge complete executing the query $A < 8$. In our example, the column is already partitioned around three pivot points 8, 10, and 14. Since the appends vector contains element 6 (i.e., an element that qualifies the query), the whole appends vector is merged. The first step of the merge is to resize our cracker column to *cracker_column.size() + appends.size()*, followed by a copy of the appends elements to the end of the column and the deletion of the appends vector. Then we must swap the newly added elements that are in the wrong piece to their correct piece. In this case, elements 6, 8, and 11 are swapped with elements in the current piece's border with the last piece. After performing the swaps, we update the cracker index pointer for 14 to point at the correct place, considering the newly inserted elements. This process is repeated until all inserted elements are placed in the correct pieces. In our example, we perform 6 swaps, and we update all 3 nodes of

[1]Our implementations and benchmarks are available at `https://github.com/pdet/ProgressiveMergesort`

our cracker index. At the end of the execution, the appends list is empty.

2.2 Merge Gradual (MG)

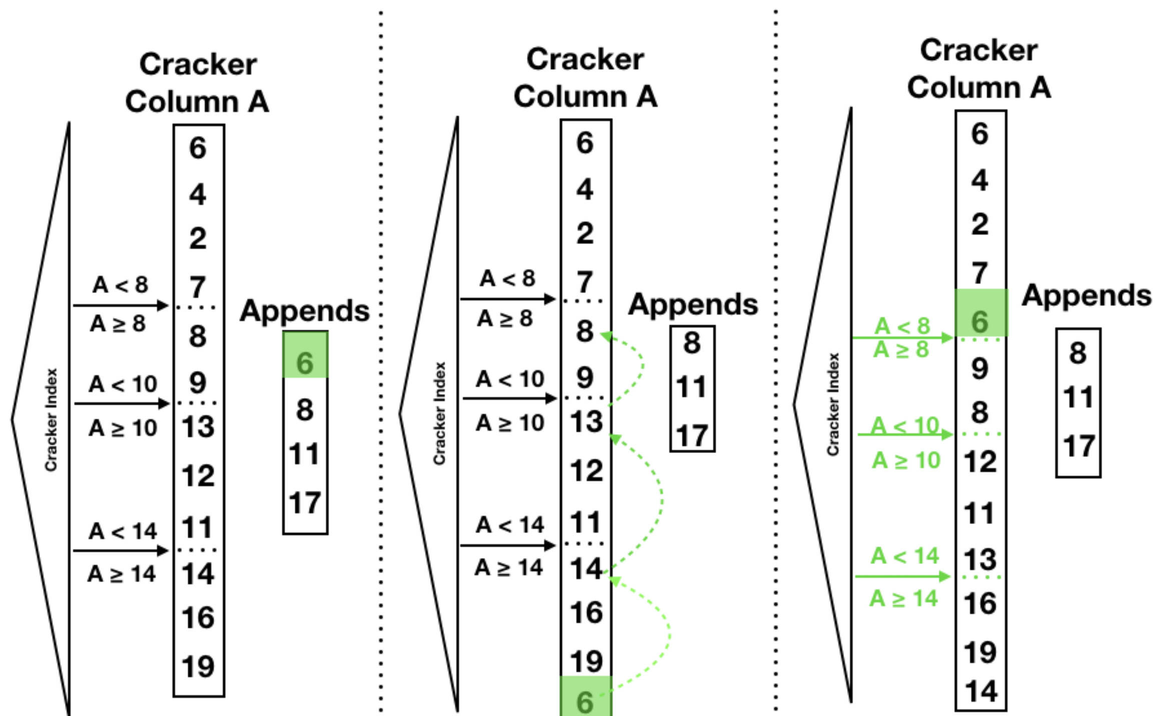


Figure 5-2: Merge Gradual on query A ¡ 8

Merge Gradual differs from Merge Complete concerning the amount of data merged per query. It only merges elements that qualify for the currently executing query.

Figure 5-2 presents the algorithm executing the $A < 8$ query in the same cracker column as before. A binary search using the query predicates is performed in the Appends vector. The elements that qualify for the query, in this case only the value 6, are merged to the cracker index. As before, value 6 is initially placed at the end of the cracker column and erased from the appends vector. Value 6 is then swapped until it reaches its correct piece, with the nodes in the cracker index being updated accordingly. Note that 3 swaps are done in this case, all 3 nodes from the cracker index are updated, and 25% of the values in the appends vector are merged.

2.3 Merge Ripple (MR)

Merge Ripple (MR) Like Merge Complete, the Merge Ripple algorithm only merges the elements that qualify for the query predicates. They differ on how they merge them. In the Merge Ripple, instead of resizing the Cracker Column and appending

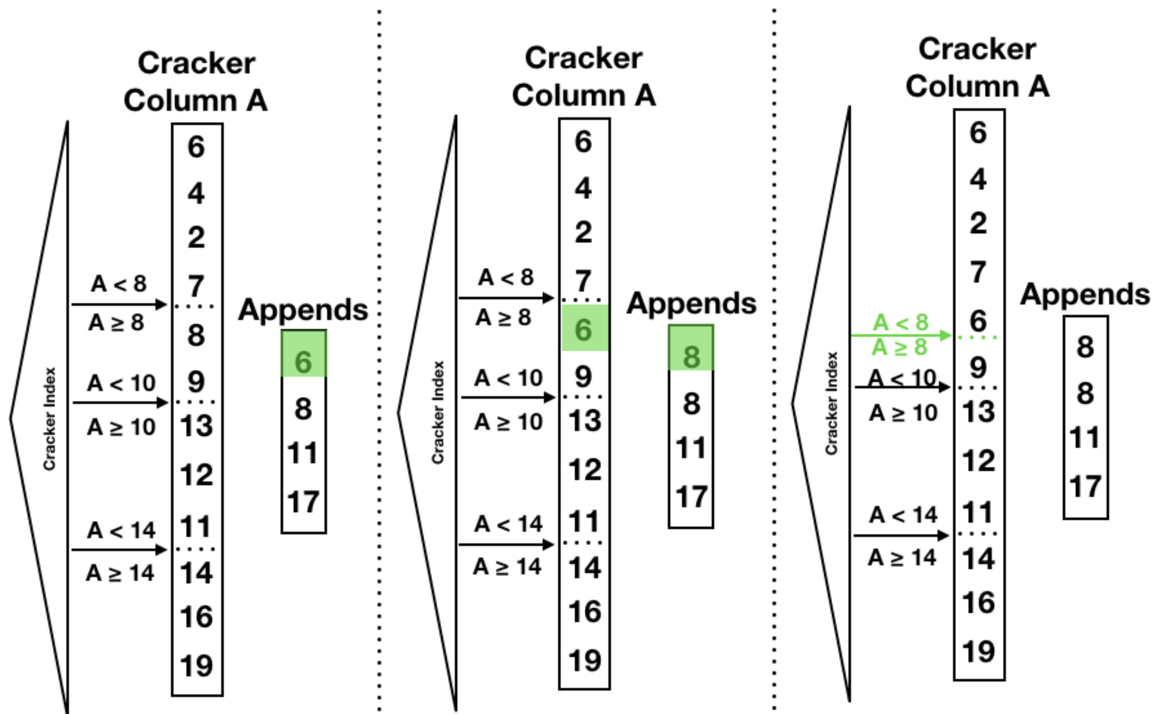


Figure 5-3: Merge Ripple on query A ¡ 8

the element to its end as its first step, it starts by swapping the to-be inserted element with the first element in the next greater-neighboring piece from its correct piece.

Figure 5-3 depicts an example of Merge Ripple executing the query $A < 8$. In our example, the column is already partitioned around three pivot points (8, 10, 14), and the appends array contains four values (6, 8, 11, 17). Since we only need to insert element 6 from the appends array, we perform a cracker index lookup and identify the element's piece (i.e., the first piece holding 6, 4, 2, and 7). We then go to the successor piece (i.e., piece 2 with elements 8 and 9) and swap the first element of that piece (8) with the element in our appends (6). After that, we only need to update the cracker index node that points to the value 8. In this case, we only had to perform 1 swap and update 1 node in the cracker. However, our append list remains with the same size it had at the start of the algorithm. Merge Ripple performs fewer swaps and updates than the previous algorithms while merging the necessary amount of data to our index.

Discussion. The Merge Complete algorithm presents the highest convergence since it fully merges the appends list whenever the appends vector has elements that qualify for the query. However, it will potentially present high-performance spikes when performing such merges. The Merge Ripple is expected to present lower performance spikes since it only merges what is necessary, avoiding column resizes,

swaps, and index node updates. However, it also presents a slow convergence and can present large performance spikes when the workload shifts to a piece where many elements must be merged. The Merge Gradual seems to be the best balance between robustness and convergence, but robustness issues similar to the Merge Ripple are still expected. Another major problem of these algorithms is the necessity of having a fully sorted appends list to merge the data efficiently. In the original paper, only small batches were used in the experiments. However, when facing large appends, the necessary a-priori sort of the append list will present a major performance bottleneck.

3 Progressive Mergesort

Progressive Mergesort is a Progressive Indexing technique inspired by the mergesort algorithm [17] and used for merging appends into the main Progressive Indexing structure. It follows the three pillars of progressive indexes: (1) low impact on query execution, (2) robust performance, and (3) guaranteed convergence. It relies on an index-budget δ that represents the percentage of the indexed per-query data, guaranteeing that the same amount of effort will be distributed for the entire workload.

In practice, during query execution, the δ defined for our Progressive Indexing algorithm is used for both the main index structure and Progressive Mergesort.

Progressive Mergesort follows two distinct canonical phases, the refinement phase and the merge phase described in this section.

Refinement. In the refinement phase, we can use any of the other proposed Progressive Indexing algorithms, getting the most performance depending on data distribution and workload. Our budget is used as described in chapter 3 depending on the algorithm executing the refinement. In this work, we decided to experiment with Progressive Quicksort as our algorithm of choice. Utilizing the other algorithms is left as an engineering exercise for future work.

Merge. At the end of the refinement phase of any Progressive Indexing algorithm, the result is a sorted list. When all merge chunks are fully sorted, we progressively merge them into one sorted chunk. We perform a progressive two-way merge in order to merge these chunks.

Figure 5-4 depicts a high-level concept of Progressive Mergesort. In this figure, red vectors are completely unsorted vectors, yellow are partially sorted vectors, and green are completely sorted. We start with our main index structure only partially sorted and with a new batch of appends.

It starts with the refinement phase. At this step, any Progressive Indexing technique

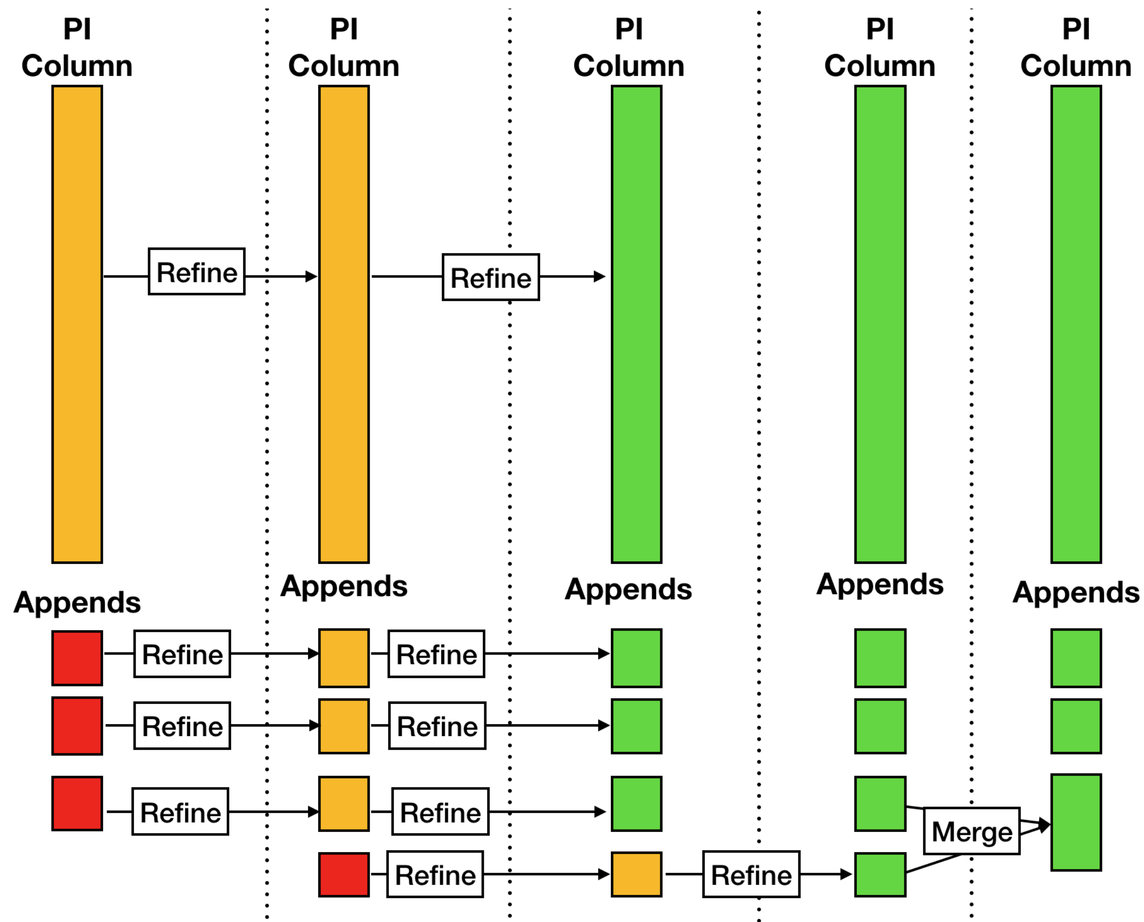


Figure 5-4: Progressive Mergesort

can be used and will continue their execution until reaching completely sorted lists. When all chunks are entirely sorted, the second phase of Progressive Mergesort starts. Here, the *Appends* arrays are progressively merged into one array. One might note that new batches can be introduced while other batches are already being refined. In this case, a Progressive Mergesort run will be initiated to newly appended chunks. All these chunks use the same δ as our main progressive index but normalized to the chunk size. Only when the original Progressive Indexing column and the appends are fully sorted (i.e., we have one sorted column for the Progressive Indexing and one sorted column for all the appends) and the appends have the same or bigger size as the Progressive Indexing column we merge them.

Figure 5-5 depicts an example of Progressive Mergesort with $delta = 0.5$. We start with two batches of updates. In the initial iterations, we execute Progressive Quicksort as the refinement phase. In *Refine (1)*, a Progressive Quicksort iteration is initiated for each chunk, since $\delta = 0.5$ both iterations index half of each chunk around one pivot. In *Refine (3)* both Progressive Quicksort iterations ended, and both chunks are fully

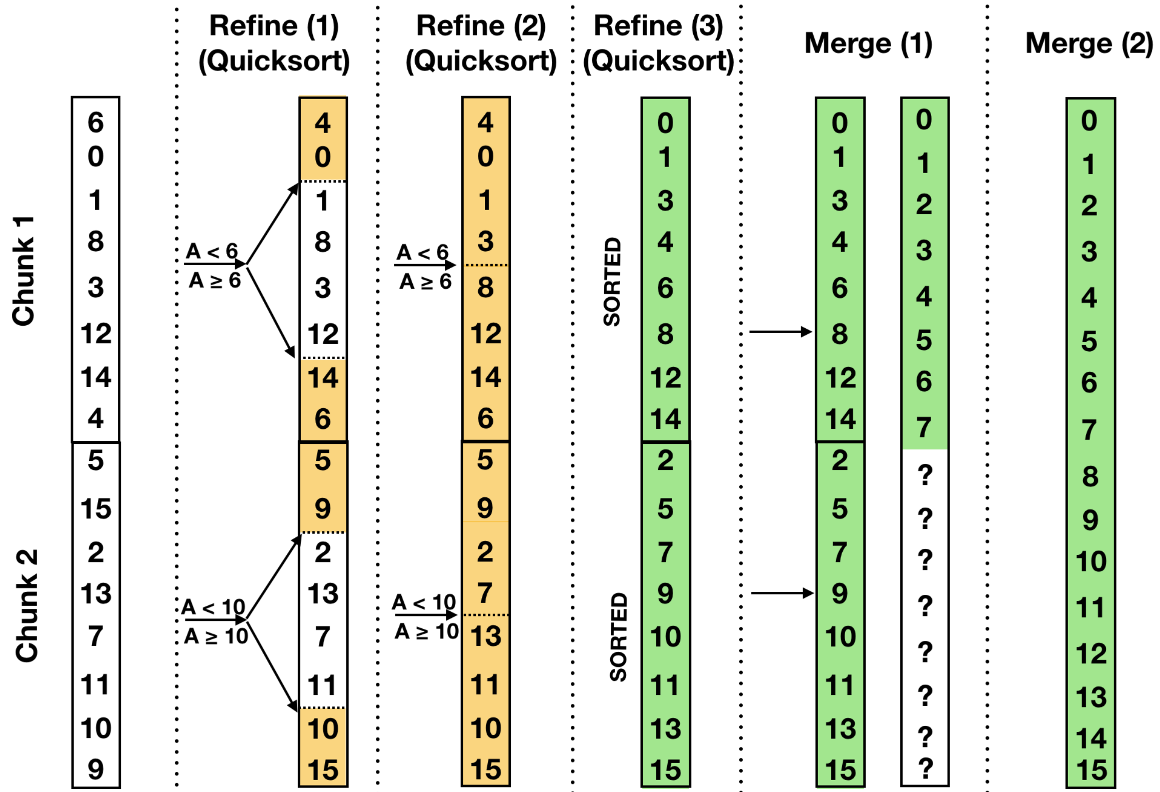


Figure 5-5: Progressive Mergesort Example ($\delta = 0.5$)

sorted. Hence we will start the merge phase of Progressive Mergesort in the following query. In *Merge (1)* we start to merge both lists using a two-way merge algorithm, and we stop when the resulting list is half complete due to our delta. For the chunks that are being merged, we must store the offsets where we stopped merging. Finally, in *Merge (2)* we end the merge phase with one completely sorted append list and delete the previous chunks.

Query Processing. When executing a query on a column with Progressive Indexing, we might encounter several arrays (i.e., the original Progressive Indexing column and batches of appends that started to be refined but are not yet merged) with different levels of refinement.

During the query execution, each array must be checked to return the elements that fit the query predicates. If the array is already fully sorted, a binary search will be executed to return the result. Otherwise, the array will be at some step of the refinement phase. Hence a lookup on the binary tree is necessary to return the offsets that match the query predicates.

When to Merge. In this work, we decided to first completely merge all appends into one, fully sorted, append array. If this array has a size equal to or bigger than the

current Progressive Indexing column, we merge both. This decision was made to avoid frequent resizes of large arrays (e.g., if we merged the Progressive Indexing column with every append first, this would result in a resize for the progressive column at every batch, which would be prohibitively expensive).

However, this decision is not necessarily optimal for all workloads. Having multiple arrays increase the random access to respond to the workload while diminishing the merge costs creating a trade-off depending on when and how these merges are performed. Creating an algorithm that decides when is the appropriate moment to merge these different arrays and which arrays should be merged is out of this chapter's scope, and we leave it as future work.

Listing 5 depicts a C++ like implementation of Progressive Mergesort. The Progressive Mergesort has as its input a vector of columns representing the chunks that are being refined, a Column representing the current set of updates, a double with the delta, the query predicates, the result structure, a pointer to the merge column, and a parameter indicating the minimum size the update column must have before entering the refinement phase. In the first *for* loop (lines 5-11), we iterate through all chunks and execute the query on each chunk. On line 6, we normalize our delta to the size of the chunk. Line 7 executes a Progressive Quicksort call that refines and returns the filtered elements of that chunk. These elements are then merged into our result structure. While checking each chunk, we also check if they are all sorted since we only start the merge phase after all chunks are already sorted.

In the second *for* loop (lines 12-15), we check if any of the elements in our current update column qualifies for the range query. If so, we add it to the result structure.

In the first *if* (lines 16-20), we initiate a merge of the two last chunks in our vector if no merge is currently happening and all chunks are sorted. The second *if* (lines 21-29) performs the actual merge, we calculate a normalized budget for the size of the *merge_column* and progressively build it. Lines 33-37 check if the merge is already finished. If it is already done, we delete the merged chunks from our chunk vector and add the newly merged chunk to the vector. We also set the pointer to the *merge_column* to null to indicate that we can initiate other merges.

The final *if* (lines 30-34) check if the *updates* column has reached a size bigger than the minimum necessary for it to become a chunk. If so, we initiate a Progressive Quicksort refinement that will be refined in the following queries. We add it to our chunk vector and create a new update column to hold the next appends.

Listing 5 Progressive Mergesort Body

```
void progressive_mergesort(vector<Column>& chunks, Column& updates,
    double delta,Query query,Result& result,MergeColumn* merge_column,
    size_t min_update_size){
    bool all_sorted = true;
    for (auto& c: chunks){
        auto budget = c.size()*delta;
        result.merge(chunk->execute(query,budget));
        if (!c.sorted){
            all_sorted = false;
        }
    }
    for (auto&u:updates){
        int match = query.match(u);
        result.maybe_push_back(u, match);
    }
    if (!merge_column && all_sorted && chunks.size() > 1){
        auto l = chunks.size() - 2;
        auto r = chunks.size() - 1;
        merge_column = new MergeColumn(chunks[l],chunks[r]);
    }
    if (merge_column){
        auto budget = merge_column.size()*delta;
        merge_column.merge(budget);
            if (merge_column.finished()){
                chunks.erase(chunks.begin()+l,chunks.begin()+r+1);
                chunks.insert(chunks.begin,merge_column);
                merge_column = nullptr;
            }
    }
    if (updates->size > min_update_size){
        auto pq = new ProgressiveQuicksort(updates);
        chunks.push_back(pq);
        updates = new Column();
    }
}
```

4 Experimental Analysis

This section provides an experimental evaluation of Progressive Mergesort and compares it with the Adaptive Merges techniques.

4.1 Setup

We implemented the Progressive Mergesort algorithm and the Adaptive Merges in a stand-alone program written in C++. The Progressive Mergesort uses Progressive Quicksort in its refinement phase.

Compilation. This application was compiled with GNU `g++` version 7.2.1 using optimization level `-O3`.

Machine. All experiments were conducted on a machine equipped with 256 GB main memory and an 8-core Intel Xeon E5-2650 v2 CPU @ 2.6 GHz with 20480 KB L3 cache.

Appends. All experiments have three parameters regarding the appends, (1) the *batch_size* that represents the size of a batch of appends, (2) the *frequency* which represents an interval of queries where a new batch of appends is executed, and (3) *start_after* that describes how many queries need to be executed before the first append happens. With these three parameters we calculate the number of appends that will be executed $total_appends = \frac{total_queries - start_after}{frequency} * batch_size$, and divide our data set into the *original_column* set that represents our initially loaded column and the *appends* set that represent the appends that will be inserted.

Data set. We generate a synthetic data set composed of $N + total_appends$ unique 8-byte integers, with $N \in \{10^7, 10^8, 10^9\}$ and representing the original column size. After generating the data set, we shuffle it following a uniform-random distribution and divide it into our original column and a list of appends.

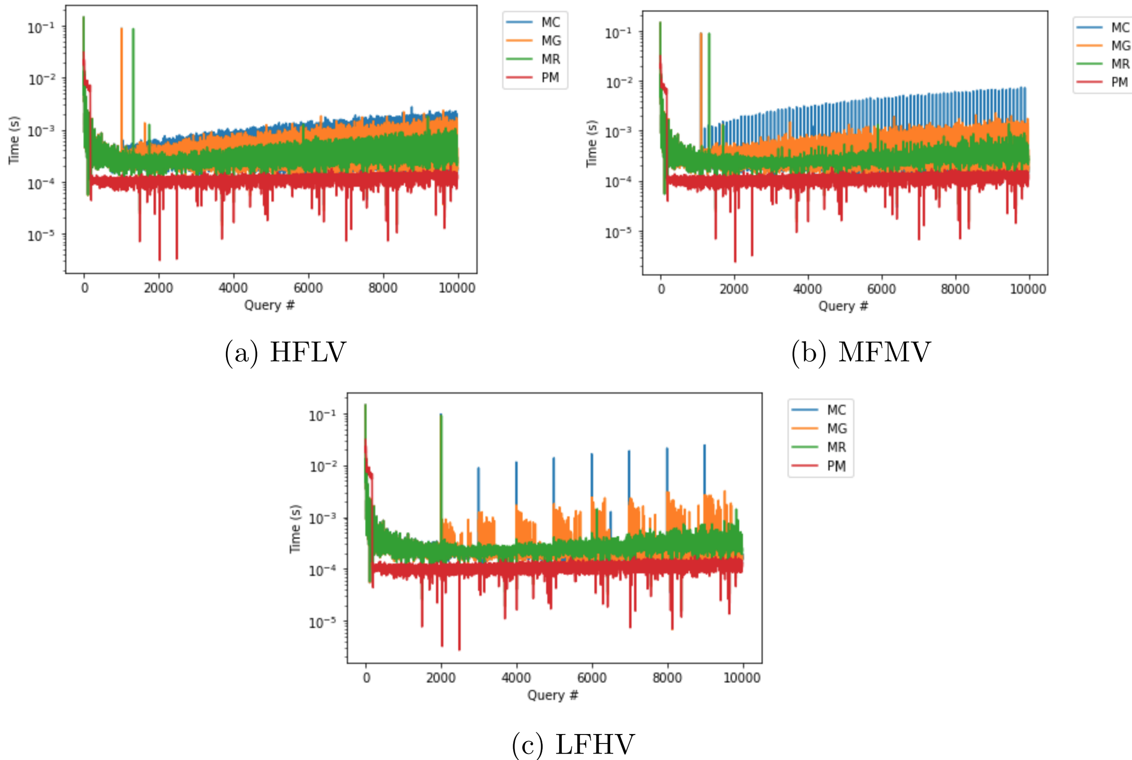
Workload. Unless stated otherwise, all experiments consist of a synthetic workload with 10^4 queries in the form `SELECT SUM(R.A) FROM R WHERE R.A BETWEEN` V_1 `AND` V_2. A random value is selected for V_1 and $V_2 = V_1 + (N + total_appends) * 1\%$.

Configuration. We experiment with 3 main configurations.

- High Frequency Low Volume (HFLV): A batch of appends with $batch_size = 0.001\% * N$ executed every 10 queries.
- Medium Frequency Medium Volume (MFMV): A batch of appends with $batch_size = 0.01\% * N$ executed every 100 queries.
- Low Frequency High Volume (LFHV): A batch of appends with $batch_size = 0.1\% * N$ executed every 1000 queries.

4.2 Performance Comparison

In this work, we decided to use the Adaptive Merges algorithms only with Adaptive Indexing due to the increased complexity of implementing them to work with Progressive Indexing and leave this task as an engineering exercise for future work. Since the base indexing algorithm is different for the Adaptive Merges and Progressive Mergesort, we decided to start appending data after 1000 queries to have refined indexes and better isolate the actual append cost from early index creation. Hence we avoid the noise of partitioning the *original_column* and focus on the actual merges from the appends. Our Progressive Mergesort uses a fixed δ of 0.1 in all experiments.



(a) HFLV

(b) MFMV

(c) LFHV

Figure 5-6: Progressive Mergesort and Adaptive Merges ($N = 10^7$ and $start_after = 1000$)

Figure 5-6 depicts a per-query performance comparison of Progressive Mergesort and Adaptive Merges. This experiment uses a data set with $N = 10^7$ and runs all three configurations described in the previous section. We continue this section by describing two observations present in all experiments, (1) regarding the column resizes and (2) an overall query robustness analysis.

Resizes. In all three configurations, HFLV, MFMV, and LFHV, we can notice that all three Adaptive Merges present a performance spike right after the start of

the updates around query 1000. The main reason for this spike is the need to resize the *Cracker Column* when appending new data. Since this resize reserves two times the space of the original *Cracker Column*, it only happens once. It is also possible to notice that with Merge Ripple, the spike occurs 100 queries later than Merge Complete and Merge Gradual. This is because Merge Ripple avoids resizing the *Cracker Column* by swapping the data from the *Appends* and the column with the actual resize only happening when we are in the last piece. This problem does not exist with Progressive Mergesort since we perform a *vector.reserve()* to allocate memory to the merge vector, and filling out the merge vector is completed over multiple queries.

Robustness. The Merge Complete presents the lowest robustness from all algorithms. Whenever a merge happens, it has a big spike upwards since it completely merges it. Merge Gradual is the second-worst. Since it completely merges all elements that qualify the predicate, it does not have one big performance spike, spreading those merges through many queries. This is particularly visible in Figure 5-6c that depicts the low-frequency high volume experiment (i.e., at every 1000 queries, a batch of size 10^4 is inserted. One can see that at every 1000 queries, there is an upwards spike that slowly decreases for 500 queries and then has a slop down since most of the *Appends* array was merged by that point. From the Adaptive Merges, the Merge Ripple presents the least variance. All queries slightly increase their cost with increasing updates. Finally, the Progressive Mergesort presents the lowest variance, with no performance spikes up.

One can notice that all algorithms present spikes downwards at the same queries overall three configurations. These are caused by noise due to the way we select our query predicates to fix our workload selectivity. Since we create our second query predicate as $V_2 = V_1 + (N + total_appends) * 1\%$. Queries might not have exactly 1% selectivity if the data is not completely merged in the column. Since the figures are with the y-axis in log scale, small differences in the selectivity produce these downwards performance spikes.

4.3 Varying Data Sizes

Table 5.1 depicts the total execution cost for the workload, excluding the initial 1000 queries. On all experiments, Progressive Mergesort presented approximately 2x better performance than the best performing Adaptive Merge algorithm. The main reason for this performance difference is that all Merge Adaptive algorithms must keep the appends sorted to merge them efficiently. This problem impacts Merge Ripple the

	Workload	MC	MG	MR	PM
10^7	HFLV	2.72	3.52	2.57	1.07
	MFMV	2.18	3.39	2.45	1.07
	LFHV	2.00	2.55	2.34	1.06
10^8	HFLV	22.76	26.16	26.61	10.64
	MFMV	20.25	26.14	25.19	10.72
	LFHV	22.14	22.42	23.89	10.63
10^9	HFLV	209.25	221.67	295.39	104.77
	MFMV	206.39	219.39	267.94	104.96
	LFHV	197.89	200.62	250.62	103.95

Table 5.1: Cumulative Time (s)

most since it tends to keep a larger appends array due to its lazier merging property. That means that a larger array must be re-sorted at every append insertion. One might notice that the results of Adaptive Merges seem to directly contradict Idreos et al. [34], where Merge Ripple was the best performing algorithm of the three. The HFLV with $N = 10^7$ is the only experiment with the same parameters as the original paper and showcases a similar result, with Merge Ripple being the fastest of the Adaptive Merges. However, as discussed before, with larger appends Merge Ripple starts to lose its benefit of fewer swaps to keep the append vector sorted.

One other interesting result is the variance in the total cost depending on the configuration of the workload. The Adaptive Merges algorithms present a much higher variance than Progressive Mergesort for the same data size. This is more prominent with larger data sizes. Taking $N = 10^9$ as an example, Merge Complete presents a variance of 11.36s, Merge Gradual of 21.05s, Merge Ripple of 44.72s, and Progressive Mergesort of 1.01s.

Compared to the Adaptive Merges algorithms, Progressive Mergesort has a very low variance from configurations at the same data size. This is due to the Progressive Mergesort algorithm not performing a complete sort in the append list but rather properly refining and merging it depending on their data size.

Table 5.2 depicts the order of magnitude of each workload's query variance on all 3 data sizes. We only calculate the query variance after executing the first 1000 queries. Note that the lower the variance, the more robust the algorithm is. As expected, Merge Complete presents the lowest robustness since it completely merges the *Appends* array to the *Cracker Column* causing a huge performance spike. The Merge Gradual and Merge Ripple are better than the Merge Complete since they only merge tuples that qualify the query predicates. Progressive Mergesort presents the highest robustness due to its indexing budget, effectively offering more fine-grained

	Workload	MC	MG	MR	PM
10^7	HFLV	e-07	e-07	e-07	e-10
	MFMV	e-06	e-07	e-07	e-10
	LFHV	e-06	e-07	e-07	e-10
10^8	HFLV	e-05	e-05	e-05	e-07
	MFMV	e-05	e-05	e-05	e-07
	LFHV	e-04	e-05	e-05	e-07
10^9	HFLV	e-03	e-03	e-03	e-06
	MFMV	e-03	e-03	e-03	e-06
	LFHV	e-02	e-03	e-03	e-06

Table 5.2: Robustness (Orders of Magnitude)

control over the stream of queries.

4.4 Appends during Index Creation

To perform a fair comparison of the Adaptive Merges and Progressive Mergesort, we only initiated the updates after 1000 queries to minimize the initial index creation cost of Adaptive Indexing and Progressive Indexing. However, after 1000 queries, the Progressive Indexing is already fully converged (i.e., the main index is a sorted list).

In this experiment, we want to evaluate Progressive Mergesort's impact during Progressive Indexing's creation phase (i.e., Initialization and Refinement phases). In our setup, we use a dataset with $N = 10^7$, a workload with 1% selectivity and 200 queries, and three different update setups. All update setups start at the first query and perform appends at every ten queries. They differ on the batches' size, with batches of size 100, 1000, and 10000.

Figure 5-7 depicts the per-query cost for the 200 queries. The height of the performance spikes are strongly correlated to the batch sizes, with larger batches introducing a higher spike. This happens due to our strategy using a fixed delta (i.e., a % of the total size of the data that is indexed per-query) for the entire workload. Hence the more data we ingest, the actual per-query cost will increase since the data size increases. One way of minimizing this issue is to extend the cost models proposed in chapter 3 to automatically generate a value for δ to reduce query variance. We leave that algorithm as an exercise for future work.

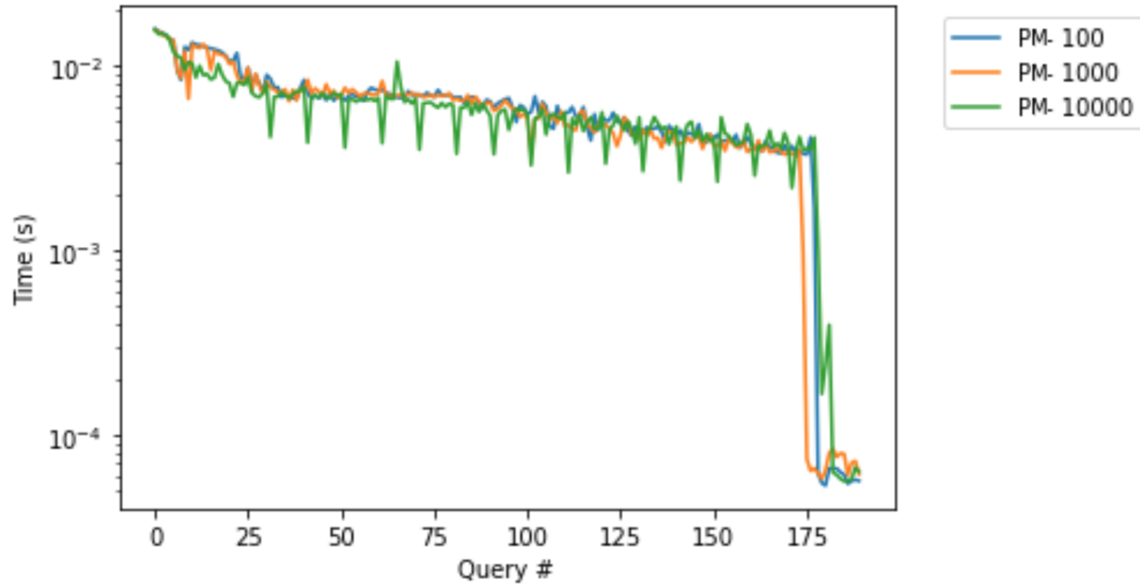


Figure 5-7: Progressive Mergesort before index convergence.

5 Summary

This chapter introduces the *Progressive Mergesort*, a novel progressive algorithm used to merge batches of appends. We compare it to the state-of-the-art merging algorithms from adaptive indexing techniques and show how they perform under multiple synthetic benchmarks. Our solution is more robust and faster than the state-of-the-art.

CHAPTER 6

Big Picture

This chapter discusses the main challenges of implementing Progressive Indexing in a database system and points out future work for the general area of incremental indexes. We also dive in specifically on unidimensional Progressive Indexing, multidimensional Progressive Indexing, and Progressive Merges.

1 The Elephant In The Room

The fact of the matter is, no database system took into production Adaptive Indexing, even though the first paper of Adaptive Indexing, *Cracking the Database Store*[38] dates from 2005. Some of the reasons, like unpredictable query response times, high penalty over initial queries, and lack of full index convergence, have been mitigated with the Progressive Indexing approach. However, other issues permeate adaptive and progressive indexes that make it unlikely for them to be picked up by a production-ready database system.

Tuple Reconstruction In most of the Adaptive/Progressive Indexing experiments, the columns must be grouped in advance when constructing the index structure, which leads to a lack of usability of the index. At the same time, real-life queries tend to filter and project over different groups of columns. For the unidimensional adaptive/progressive index, this problem is obvious. Only one column is indexed when selecting any other column. We must perform tuple reconstruction, which hides any potential benefit from having the data skipping from the index, except for point

queries and high selective queries that would not be classified as Analytical Processing. Multidimensional Adaptive/Progressive Indexing presents the same problem since it only groups the data if they have filters. Hence selections on columns that are not being filtered would have to perform tuple reconstruction. One way of mitigating the tuple reconstruction would be to create the index by not only copying the filtered columns but the whole table. Of course, this presents problems in itself since it will cause a storage blow-up (i.e., now every index must own a copy of the full table) and increase the updates' costs.

Overhead of Storage/Maintenance Every query with a filter will produce either a unidimensional or a multidimensional Adaptive/Progressive Indexing, depending on the number of filters the query has. In the worst case, at some point, every column will have a unidimensional index created, and one multidimensional index will be created for every unique combination of multidimensional filters. This, of course, will cause a storage blow-up and a maintenance overhead that will make it impossible to use these techniques on a real exploratory dataset.

Is there hope? We believe that the next step to the grand area of Adaptive/Progressive Indexes is to move from secondary index creation to Adaptive/Progressive Table Partitioning. The basic idea is to perform the partitioning used to create indexes and reorganize the table's data instead of creating a secondary index structure. This would increase the usability of the data reorganization since the multidimensional indexes will suffer from tuple reconstruction costs when accessing non-indexed tuples.

2 Future Work

In this section, we will present potential future research directions in the area of Progressive Indexes. We split up this section by Progressive Indexes and Progressive Merges.

2.1 Progressive Indexes

We point out the following as the main aspects to be explored in Progressive Indexes future work:

- **Approximate Query Processing.** One could also resort to using approximate query processing techniques [12] to allow for a faster convergence (i.e., by spending less time scanning data for the query, we can invest more time indexing data). We can then build a progressive index as a by-product of the approximate

query processing, leading to better accuracy and faster responses as the data is queried more often.

- **Indexing Methods.** Other techniques can be adapted to work progressively with different benefits. For example, instead of constructing the complete hash table, we only insert $n * \delta$ elements and scan the column's remainder. The partial hash table can be used to answer point queries on the indexed part of the data. Another example is column imprints [54] where instead of immediately building imprints for the entire column, only build them for the first fraction δ of the data.

- **Interleaving Progressive Strategies.** As depicted in our decision tree, different progressive strategies can be more efficient in different scenarios. When the indexing budget is small, the indexes can take longer to converge fully. This longer period increases the chances of sudden changes in the workload patterns before the index is fully built. Detecting these changes and changing the progressive strategy on the fly can be beneficial for these cases.

- **Indexing Structures.** Different data structures can be used to exploit modern hardware and boost access to more selective queries. In chapter 3, we choose to progressively build a B+-Tree in our consolidation phase. However, other structures like the ART-tree [40] can also be built progressively, with more careful considerations on their creation costs and query performance.

- **Complex Database Operations.** Much like regular indexes, progressive indexes could also be used for other database operations such as joins and aggregations.

2.2 Progressive Merges

In chapter 5 we introduce a novel algorithm for merging appends into progressive indexes. The work has still several engineering and research steps that must be taken as future work:

- **Integrating Merge Ripple With Progressive Indexing.** In our experiments, we compare against adaptive indexing using the merge gradual/complete/ripple algorithms. However, this comparison would be even more significant if these algorithms were implemented directly into Progressive Indexing. For

example, if the main index algorithm is Progressive Quicksort, by using an AVL-Tree, similar merge algorithms could be used.

- **Refinement Method.** In chapter 5, we only use Progressive Quicksort as our refinement strategy within Progressive Mergesort. However, in the Progressive Indexing work, it is demonstrated that different Progressive Indexing algorithms can present better performance depending on the data distribution and workload. With mergesort, we can select a different algorithm for each chunk in the refinement step. Deciding which algorithm to use could drastically improve performance.

- **Merge Strategy.** Deciding when to merge and which arrays to merge can be beneficial to the cumulative cost of the workload since there is a trade-off on the random access versus merging costs (i.e., keeping many smaller arrays or frequently merging them in order to maintain only a small number of bigger arrays). An algorithm that takes that this trade-off into consideration is left as future work.

- **Greedy Progressive Mergesort.** Our current implementation of Progressive Mergesort relies on a fixed δ for the entire workload. The development of a cost-model to the merge phase will integrate it with greedy Progressive Indexing algorithms. Hence, as future work, a greedy version of our Progressive Mergesort can bring even fewer performance spikes to our algorithm.

- **Handling Updates.** We describe how to efficiently merge appends since these are the most common types of updates in interactive data analysis. However, although deletes and updates are not frequent, they might still occur. Therefore Progressive Mergesort must be capable of properly handling them.

- **Multidimensional Updates.** Until now, we only focused on unidimensional Progressive Indexing. However, multidimensional Progressive Indexing [43] was recently proposed to efficiently index columns for queries with multiple selective filters. In this algorithm, a KD-Tree is used to store and navigate the partitions created by Progressive Indexing. To support updates on this structure, Progressive Mergesort must be extended to consider the KD-Tree nodes to merge multiple batches of updates correctly.

- **Real Benchmarks.** The Sloan Digital Sky Survey [1] is an open-source project

[1] `https://www.sdss.org/`

that maps the universe with an open data set and interactive-exploratory query logs. Capturing the updates on this database can represent real patterns of updates on interactive data.

Summary

Interactive exploration of large volumes of data is increasingly common, as data scientists attempt to extract interesting information from large opaque data sets. This scenario presents a difficult challenge for traditional database systems, as (1) nothing is known about the query workload in advance, (2) the query workload is constantly changing, and (3) the system must provide interactive responses to the issued queries. This environment is challenging for index creation, as traditional database indexes require upfront creation, hence a priori workload knowledge, to be efficient.

In this work, we introduce *Progressive Indexing*, a novel performance-driven indexing technique that focuses on automatic index creation while providing interactive response times to incoming queries. Its design allows queries to have a limited budget to spend on index creation. The indexing budget is automatically tuned to each query before query processing. This allows for systems to provide interactive answers to queries during index creation while being robust against various workload patterns and data distributions.

We develop progressive algorithms to index one and multiple dimensions. In addition, we introduce *Progressive Merges*, a robust algorithm that merges appends into our Progressive Indexes without penalizing single queries.

Samenvatting

Interactieve verkenning van grote hoeveelheden gegevens komt steeds vaker voor, omdat datawetenschappers proberen interessante informatie te extraheren uit grote complexe gegevenssets. Dit scenario vormt een uitdaging voor traditionele databasesystemen, aangezien (1) er van tevoren niets bekend is over de query-workload, (2) de query-workload voortdurend verandert, en (3) het systeem interactieve antwoorden moet geven op de uitgegeven queries. Deze omgeving is een uitdaging voor het maken van indexen, aangezien traditionele database-indexen vooraf moeten worden gemaakt, en dus a priori kennis van de werkbelasting nodig hebben, om efficiënt te zijn.

In dit werk introduceren we *Progressive Indexing*, een nieuwe prestatiegerichte indexeringstechniek die zich richt op het automatisch bouwen van indexen en tegelijkertijd interactieve reactietijden biedt op inkomende vragen. Dankzij het ontwerp kunnen zoekopdrachten een beperkt budget hebben om te besteden aan het maken van indexen. Het indexeringsbudget wordt automatisch afgestemd op elke query voordat de query wordt verwerkt. Hierdoor kunnen systemen interactieve antwoorden geven op vragen tijdens het maken van een index, terwijl ze robuust zijn tegen verschillende werkbelastingpatronen en gegevensverdelingen.

We ontwikkelen progressieve algoritmen om één en meerdere dimensies te indexeren. Daarnaast introduceren we *Progressive Merges*, een robuust algoritme dat toevoegingen in onze progressieve indexen samenvoegt zonder afzonderlijke zoekopdrachten te bestraffen.

Publications

This thesis is based on the following set of publications:

- **Progressive Mergesort: Merging Batches of Appends into Progressive Indexes**, Pedro Holanda and Stefan Manegold, 24th International Conference on Extending Database Technology (EDBT 2021)
- **Multidimensional Adaptive & Progressive Indexes**, Matheus Nerone, Pedro Holanda, Eduardo Almeida and Stefan Manegold, 37th International Conference on Data Engineering (ICDE 2021)
- **Progressive Indexes: Indexing for Interactive Data Analysis**, Pedro Holanda, Mark Raasveldt, Stefan Manegold and Hannes Mühleisen, 46th International Conference on Very Large Data Bases (VLDB 2020)
- **Cracking KD-Tree: The First Multidimensional Adaptive Indexing.**, Pedro Holanda, Matheus Nerone, Eduardo Almeida, and Stefan Manegold, 7th International Conference on Data Science, Technology and Applications (DATA 2018, EDDY)
- **Progressive Indices – Indexing Without Prejudice.**, Pedro Holanda, 44th International Conference on Very Large Data Bases (VLDB 2018, PhD Workshop)

Further set of publications not included in this thesis:

- **Relational Queries with a Tensor Processing Unit**, Pedro Holanda and Hannes Mühleisen, ACM International Conference on Management of Data (SIGMOD 2019, DaMoN)

- **devUDF: Increasing UDF development efficiency through IDE Integration. It works like a PyCharm!**, Mark Raasveldt, Pedro Holanda and Stefan Manegold, 22nd International Conference on Extending Database Technology (EDBT 2019, Demo Track)
- **Fair Benchmarking Considered Difficult:Common Pitfalls In Database Performance Testing.**, Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen, ACM International Conference on Management of Data (SIGMOD 2018, DbTest)
- **Deep Integration of Machine Learning Into Column Stores**, Mark Raasveldt, Pedro Holanda, Hannes Mühleisen and Stefan Manegold, 21st International Conference on Extending Database Technology (EDBT 2018)
- **Don't Hold My UDFs Hostage - Exporting UDFs For Debugging Purposes**, Pedro Holanda, Mark Raasveldt and Martin Kersten, 32nd Simpósio Brasileiro de Bancos de Dados (SBBD 2017)

Curriculum Vitae

Pedro Thiago Timbó Holanda geboren op 30 July 1992 te Fortaleza/Brazilië.

2021 - Current	Post-Doc Database Architectures group Centrum van Wiskunde & Informatica (CWI) Supervised by Hannes Mühleisen
2017 - 2021	PhD Candidate Database Architectures group Centrum van Wiskunde & Informatica (CWI) Supervised by Stefan Manegold, Hannes Mühleisen and Peter Boncz
2019 - 2019	PhD Intern Data Management, Exploration and Mining group Microsoft Research Institute
2014 - 2016	Master of Science Computing Science Universidade Federal do Paraná Supervised by Eduardo C. de Almeida
2010 - 2014	Bachelor of Science Computer Science Universidade Federal do Ceará

Bibliography

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[2] V. Alvarez, S. Richter, X. Chen, and J. Dittrich. A comparison of adaptive radix trees and hash tables. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1227–1238. IEEE, 2015.

[3] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah. Aqwa: adaptive query workload aware partitioning of big spatial data. *PVLDB*, 8(13):2062–2073, 2015.

[4] L. Battle, P. Eichmann, M. Angelini, T. Catarci, G. Santucci, Y. Zheng, C. Binnig, J.-D. Fekete, and D. Moritz. Database benchmarking for supporting real-time interactive querying of large data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1571–1587, 2020.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, 1990.

[6] J. Bell and G. Gupta. An evaluation of self-adjusting binary search tree techniques. *Software: Practice and Experience*, 23(4):369–382, 1993.

[7] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[8] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis. Hot: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 521–534, 2018.

[9] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 54–65, 1999.

[10] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Cidr*, volume 5, pages 225–237, 2005.

[11] N. Bruno. *Automated Physical Database Design and Tunning*. CRC-Press, 2011.

[12] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. *The VLDB Journal—The International Journal on Very Large Data Bases*, 10(2-3):199–223, 2001.

[13] S. Chaudhuri and V. Narasayya. AutoAdmin "What-if" Index Analysis Utility. *ACM SIGMOD Record*, 27(2):367–378, 1998.

[14] S. Chaudhuri and V. R. Narasayya. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, volume 97, pages 146–155, 1997.

[15] D. Comer. The Difficulty of Optimum Index Selection. *ACM Transactions on Database Systems (TODS)*, 3(4):440–445, 1978.

[16] . G. P. Consortium et al. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[18] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[19] T. Georgiou, S. Schmitt, T. Bäck, N. Pu, W. Chen, and M. Lew. Comparison of deep learning and hand crafted features for mining simulation data. In *ICPR 2020)*. IEEE, 2020.

[20] T. Georgiou, S. Schmitt, M. Olhofer, Y. Liu, T. Bäck, and M. Lew. Learning fluid flows. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2018.

[21] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 371–381. ACM, 2010.

[22] G. Graefe and H. Kuno. Modern b-tree techniques. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1370–1373. IEEE, 2011.

[23] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index Selection for OLAP. In *Data Engineering, 1997. Proceedings. 13th International Conference on*, pages 208–219. IEEE, 1997.

[24] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[25] I. Haffner, F. M. Schuhknecht, and J. Dittrich. An Analysis and Comparison of Database Cracking Kernels. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, DAMON '18, pages 10:1–10:10, New York, NY, USA, 2018. ACM.

[26] F. Halim, S. Idreos, P. Karras, and R. H. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 5(6):502–513, 2012.

[27] C. A. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[28] P. Holanda. Progressive indices: Indexing without prejudice. In *PhD@ VLDB*, 2018.

[29] P. Holanda and E. C. de Almeida. SPST-Index: A Self-Pruning Splay Tree Index for Caching Database Cracking. In *EDBT*, pages 458–461, 2017.

[30] P. Holanda and S. Manegold. Progressive mergesort: Merging batches of appends into progressive indexes. *EDBT*, 2021.

[31] P. Holanda, M. Nerone, E. C. de Almeida, and S. Manegold. Cracking kd-tree: The first multidimensional adaptive indexing (position paper). In *DATA*, pages 393–399, 2018.

[32] P. Holanda, M. Raasveldt, S. Manegold, and H. Mühleisen. Progressive indexes: indexing for interactive data analysis. *PVLDB*, 12(13):2366–2378, 2019.

[33] HRI, LIACS, and CWI. Damioso: Data mining on high volume simulation output, 2020.

[34] S. Idreos, M. L. Kersten, and S. Manegold. Updating a Cracked Database. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 413–424, New York, NY, USA, 2007. ACM.

[35] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column-stores. *SIGMOD*, pages 297–308, 2009.

[36] S. Idreos, M. L. Kersten, S. Manegold, et al. Database Cracking. In *CIDR*, volume 3, pages 1–8, 2007.

[37] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9):586–597, 2011.

[38] M. L. Kersten, S. Manegold, et al. Cracking the database store. In *CIDR*, volume 5, pages 4–7. Citeseer, 2005.

[39] E. Leal and L. Gruenwald. A study on database cracking with gpus. *ADMS 2019*, 2019.

[40] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, pages 38–49. IEEE, 2013.

[41] Z. Liu and J. Heer. The Effects of Interactive Latency on Exploratory Visual Analysis. *Visualization and Computer Graphics, IEEE Transactions on*, 20:2122–2131, 12 2014.

[42] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. *CoRR*, 2019.

[43] M. Nerone, P. Holanda, E. C. De Almeida, and S. Manegold. Multidimensional adaptive and progressive indexes. In *IEEE International Conference on Data Engineering (ICDE)*, 2021.

[44] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, et al. Self-Driving Database Management Systems. In *CIDR*, 2017.

[45] M. Pavlovic, D. Sidlauskas, T. Heinis, and A. Ailamaki. Quasii: query-aware spatial incremental index. In *EDBT*, pages 325–336, 2018.

[46] E. Petraki, S. Idreos, and S. Manegold. Holistic Indexing in Main-memory Column-stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1153–1166. ACM, 2015.

[47] H. Pirk, E. Petraki, S. Idreos, S. Manegold, and M. Kersten. Database Cracking: Fancy Scan, not Poor Man's Sort! In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, page 4. ACM, 2014.

[48] K. A. Ross. Conjunctive selection conditions in main memory. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 109–120, 2002.

[49] F. M. Schuhknecht, J. Dittrich, and L. Linden. Adaptive adaptive indexing. *ICDE*, 2018.

[50] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *PVLDB*, 7(2):97–108, 2013.

[51] F. M. Schuhknecht, P. Khanchandani, and J. Dittrich. On the surprising difficulty of simple things: the case of radix partitioning. *PVLDB*, 8(9):934–937, 2015.

[52] T. Sellam, E. Müller, and M. Kersten. Semi-Automated Exploration of Data Warehouses. In *CIKM*, pages 1321–1330, 10 2015.

[53] A. Sharma, F. M. Schuhknecht, and J. Dittrich. The Case for Automatic Database Administration using Deep Reinforcement Learning. *arXiv preprint arXiv:1801.05643*, 2018.

[54] L. Sidirourgos and M. Kersten. Column Imprints: A Secondary Index Structure. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 893–904, New York, NY, USA, 2013. ACM.

[55] S. Sprenger, P. Schäfer, and U. Leser. Multidimensional range queries on modern hardware. In *SSDBM*, pages 4:1–4:12, 2018.

[56] A. S. Szalay, J. Gray, A. R. Thakar, P. Z. Kunszt, T. Malik, J. Raddick, C. Stoughton, and J. vandenBerg. The SDSS skyserver: public access to the sloan digital sky server data. In *SIGMOD*, pages 570–581, 2002.

[57] E. Teixeira, P. Amora, and J. C. Machado. Metisidx-from adaptive to predictive data indexing. In *EDBT*, pages 485–488, 2018.

[58] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 101–110. IEEE, 2000.

[59] S. Wang, D. Maier, and B. C. Ooi. Fast and adaptive indexing of multi-dimensional observational data. *PVLDB*, 2016.

[60] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, pages 473–488, 2018.

[61] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, volume 98, pages 194–205, 1998.

[62] F. Zardbani, P. Afshani, and P. Karras. Revisiting the theory and practice of database cracking. In *EDBT*, pages 415–418, 2020.

[63] T. Zäschke, C. Zimmerli, and M. C. Norrie. The ph-tree: a space-efficient storage structure and multi-dimensional index. In *SIGMOD*, pages 397–408, 2014.

Siks Dissertation Series

2011 01 Botond Cseke (RUN), Variational Algorithms for Bayesian Inference in Latent Gaussian Models

02 Nick Tinnemeier (UU), Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language

03 Jan Martijn van der Werf (TUE), Compositional Design and Verification of Component-Based Information Systems

04 Hado van Hasselt (UU), Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference

05 Bas van der Raadt (VU), Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.

06 Yiwen Wang (TUE), Semantically-Enhanced Recommendations in Cultural Heritage

07 Yujia Cao (UT), Multimodal Information Presentation for High Load Human Computer Interaction

08 Nieske Vergunst (UU), BDI-based Generation of Robust Task-Oriented Dialogues

09 Tim de Jong (OU), Contextualised Mobile Media for Learning

10 Bart Bogaert (UvT), Cloud Content Contention

11 Dhaval Vyas (UT), Designing for Awareness: An Experience-focused HCI Perspective

12 Carmen Bratosin (TUE), Grid Architecture for Distributed Process Mining

13 Xiaoyu Mao (UvT), Airport under Control. Multiagent Scheduling for Airport Ground Handling

14 Milan Lovric (EUR), Behavioral Finance and Agent-Based Artificial Markets

15 Marijn Koolen (UvA), The Meaning of Structure: the Value of Link Evidence for Information Retrieval
16 Maarten Schadd (UM), Selective Search in Games of Different Complexity
17 Jiyin He (UVA), Exploring Topic Structure: Coherence, Diversity and Relatedness
18 Mark Ponsen (UM), Strategic Decision-Making in complex games
19 Ellen Rusman (OU), The Mind's Eye on Personal Profiles
20 Qing Gu (VU), Guiding service-oriented software engineering - A view-based approach
21 Linda Terlouw (TUD), Modularization and Specification of Service-Oriented Systems
22 Junte Zhang (UVA), System Evaluation of Archival Description and Access
23 Wouter Weerkamp (UVA), Finding People and their Utterances in Social Media
24 Herwin van Welbergen (UT), Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior
25 Syed Waqar ul Qounain Jaffry (VU), Analysis and Validation of Models for Trust Dynamics
26 Matthijs Aart Pontier (VU), Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots
27 Aniel Bhulai (VU), Dynamic website optimization through autonomous management of design patterns
28 Rianne Kaptein (UVA), Effective Focused Retrieval by Exploiting Query Context and Document Structure
29 Faisal Kamiran (TUE), Discrimination-aware Classification
30 Egon van den Broek (UT), Affective Signal Processing (ASP): Unraveling the mystery of emotions
31 Ludo Waltman (EUR), Computational and Game-Theoretic Approaches for Modeling Bounded Rationality
32 Nees-Jan van Eck (EUR), Methodological Advances in Bibliometric Mapping of Science
33 Tom van der Weide (UU), Arguing to Motivate Decisions
34 Paolo Turrini (UU), Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations
35 Maaike Harbers (UU), Explaining Agent Behavior in Virtual Training
36 Erik van der Spek (UU), Experiments in serious game design: a cognitive approach

37 Adriana Burlutiu (RUN), Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference
38 Nyree Lemmens (UM), Bee-inspired Distributed Optimization
39 Joost Westra (UU), Organizing Adaptation using Agents in Serious Games
40 Viktor Clerc (VU), Architectural Knowledge Management in Global Software Development
41 Luan Ibraimi (UT), Cryptographically Enforced Distributed Data Access Control
42 Michal Sindlar (UU), Explaining Behavior through Mental State Attribution
43 Henk van der Schuur (UU), Process Improvement through Software Operation Knowledge
44 Boris Reuderink (UT), Robust Brain-Computer Interfaces
45 Herman Stehouwer (UvT), Statistical Language Models for Alternative Sequence Selection
46 Beibei Hu (TUD), Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work
47 Azizi Bin Ab Aziz (VU), Exploring Computational Models for Intelligent Support of Persons with Depression
48 Mark Ter Maat (UT), Response Selection and Turn-taking for a Sensitive Artificial Listening Agent
49 Andreea Niculescu (UT), Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality

---

2012 01 Terry Kakeeto (UvT), Relationship Marketing for SMEs in Uganda
02 Muhammad Umair (VU), Adaptivity, emotion, and Rationality in Human and Ambient Agent Models
03 Adam Vanya (VU), Supporting Architecture Evolution by Mining Software Repositories
04 Jurriaan Souer (UU), Development of Content Management System-based Web Applications
05 Marijn Plomp (UU), Maturing Interorganisational Information Systems
06 Wolfgang Reinhardt (OU), Awareness Support for Knowledge Workers in Research Networks
07 Rianne van Lambalgen (VU), When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions
08 Gerben de Vries (UVA), Kernel Methods for Vessel Trajectories
09 Ricardo Neisse (UT), Trust and Privacy Management Support for Context-Aware Service Platforms

10 David Smits (TUE), Towards a Generic Distributed Adaptive Hypermedia Environment
11 J.C.B. Rantham Prabhakara (TUE), Process Mining in the Large: Preprocessing, Discovery, and Diagnostics
12 Kees van der Sluijs (TUE), Model Driven Design and Data Integration in Semantic Web Information Systems
13 Suleman Shahid (UvT), Fun and Face: Exploring non-verbal expressions of emotion during playful interactions
14 Evgeny Knutov (TUE), Generic Adaptation Framework for Unifying Adaptive Web-based Systems
15 Natalie van der Wal (VU), Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes.
16 Fiemke Both (VU), Helping people by understanding them - Ambient Agents supporting task execution and depression treatment
17 Amal Elgammal (UvT), Towards a Comprehensive Framework for Business Process Compliance
18 Eltjo Poort (VU), Improving Solution Architecting Practices
19 Helen Schonenberg (TUE), What's Next? Operational Support for Business Process Execution
20 Ali Bahramisharif (RUN), Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing
21 Roberto Cornacchia (TUD), Querying Sparse Matrices for Information Retrieval
22 Thijs Vis (UvT), Intelligence, politie en veiligheidsdienst: verenigbare grootheden?
23 Christian Muehl (UT), Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction
24 Laurens van der Werff (UT), Evaluation of Noisy Transcripts for Spoken Document Retrieval
25 Silja Eckartz (UT), Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application
26 Emile de Maat (UVA), Making Sense of Legal Text
27 Hayrettin Gurkok (UT), Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games
28 Nancy Pascall (UvT), Engendering Technology Empowering Women
29 Almer Tigelaar (UT), Peer-to-Peer Information Retrieval
30 Alina Pommeranz (TUD), Designing Human-Centered Systems for Reflective Decision Making
31 Emily Bagarukayo (RUN), A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure

32 Wietske Visser (TUD), Qualitative multi-criteria preference representation and reasoning

33 Rory Sie (OUN), Coalitions in Cooperation Networks (COCOON)

34 Pavol Jancura (RUN), Evolutionary analysis in PPI networks and applications

35 Evert Haasdijk (VU), Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics

36 Denis Ssebugwawo (RUN), Analysis and Evaluation of Collaborative Modeling Processes

37 Agnes Nakakawa (RUN), A Collaboration Process for Enterprise Architecture Creation

38 Selmar Smit (VU), Parameter Tuning and Scientific Testing in Evolutionary Algorithms

39 Hassan Fatemi (UT), Risk-aware design of value and coordination networks

40 Agus Gunawan (UvT), Information Access for SMEs in Indonesia

41 Sebastian Kelle (OU), Game Design Patterns for Learning

42 Dominique Verpoorten (OU), Reflection Amplifiers in self-regulated Learning

43 Withdrawn

44 Anna Tordai (VU), On Combining Alignment Techniques

45 Benedikt Kratz (UvT), A Model and Language for Business-aware Transactions

46 Simon Carter (UVA), Exploration and Exploitation of Multilingual Data for Statistical Machine Translation

47 Manos Tsagkias (UVA), Mining Social Media: Tracking Content and Predicting Behavior

48 Jorn Bakker (TUE), Handling Abrupt Changes in Evolving Time-series Data

49 Michael Kaisers (UM), Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions

50 Steven van Kervel (TUD), Ontologogy driven Enterprise Information Systems Engineering

51 Jeroen de Jong (TUD), Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching

---

2013 01 Viorel Milea (EUR), News Analytics for Financial Decision Support

02 Erietta Liarou (CWI), MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing

03 Szymon Klarman (VU), Reasoning with Contexts in Description Logics

04 Chetan Yadati (TUD), Coordinating autonomous planning and scheduling

05 Dulce Pumareja (UT), Groupware Requirements Evolutions Patterns
06 Romulo Goncalves (CWI), The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience
07 Giel van Lankveld (UvT), Quantifying Individual Player Differences
08 Robbert-Jan Merk (VU), Making enemies: cognitive modeling for opponent agents in fighter pilot simulators
09 Fabio Gori (RUN), Metagenomic Data Analysis: Computational Methods and Applications
10 Jeewanie Jayasinghe Arachchige (UvT), A Unified Modeling Framework for Service Design.
11 Evangelos Pournaras (TUD), Multi-level Reconfigurable Self-organization in Overlay Services
12 Marian Razavian (VU), Knowledge-driven Migration to Services
13 Mohammad Safiri (UT), Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly
14 Jafar Tanha (UVA), Ensemble Approaches to Semi-Supervised Learning Learning
15 Daniel Hennes (UM), Multiagent Learning - Dynamic Games and Applications
16 Eric Kok (UU), Exploring the practical benefits of argumentation in multi-agent deliberation
17 Koen Kok (VU), The PowerMatcher: Smart Coordination for the Smart Electricity Grid
18 Jeroen Janssens (UvT), Outlier Selection and One-Class Classification
19 Renze Steenhuizen (TUD), Coordinated Multi-Agent Planning and Scheduling
20 Katja Hofmann (UvA), Fast and Reliable Online Learning to Rank for Information Retrieval
21 Sander Wubben (UvT), Text-to-text generation by monolingual machine translation
22 Tom Claassen (RUN), Causal Discovery and Logic
23 Patricio de Alencar Silva (UvT), Value Activity Monitoring
24 Haitham Bou Ammar (UM), Automated Transfer in Reinforcement Learning
25 Agnieszka Anna Latoszek-Berendsen (UM), Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System
26 Alireza Zarghami (UT), Architectural Support for Dynamic Homecare Service Provisioning

27 Mohammad Huq (UT), Inference-based Framework Managing Data Provenance
28 Frans van der Sluis (UT), When Complexity becomes Interesting: An Inquiry into the Information eXperience
29 Iwan de Kok (UT), Listening Heads
30 Joyce Nakatumba (TUE), Resource-Aware Business Process Management: Analysis and Support
31 Dinh Khoa Nguyen (UvT), Blueprint Model and Language for Engineering Cloud Applications
32 Kamakshi Rajagopal (OUN), Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development
33 Qi Gao (TUD), User Modeling and Personalization in the Microblogging Sphere
34 Kien Tjin-Kam-Jet (UT), Distributed Deep Web Search
35 Abdallah El Ali (UvA), Minimal Mobile Human Computer Interaction
36 Than Lam Hoang (TUe), Pattern Mining in Data Streams
37 Dirk Börner (OUN), Ambient Learning Displays
38 Eelco den Heijer (VU), Autonomous Evolutionary Art
39 Joop de Jong (TUD), A Method for Enterprise Ontology based Design of Enterprise Information Systems
40 Pim Nijssen (UM), Monte-Carlo Tree Search for Multi-Player Games
41 Jochem Liem (UVA), Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning
42 Léon Planken (TUD), Algorithms for Simple Temporal Reasoning
43 Marc Bron (UVA), Exploration and Contextualization through Interaction and Concepts

---

2014 01 Nicola Barile (UU), Studies in Learning Monotone Models from Data
02 Fiona Tuliyano (RUN), Combining System Dynamics with a Domain Modeling Method
03 Sergio Raul Duarte Torres (UT), Information Retrieval for Children: Search Behavior and Solutions
04 Hanna Jochmann-Mannak (UT), Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation
05 Jurriaan van Reijsen (UU), Knowledge Perspectives on Advancing Dynamic Capability
06 Damian Tamburri (VU), Supporting Networked Software Development
07 Arya Adriansyah (TUE), Aligning Observed and Modeled Behavior

08 Samur Araujo (TUD), Data Integration over Distributed and Heterogeneous Data Endpoints
09 Philip Jackson (UvT), Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language
10 Ivan Salvador Razo Zapata (VU), Service Value Networks
11 Janneke van der Zwaan (TUD), An Empathic Virtual Buddy for Social Support
12 Willem van Willigen (VU), Look Ma, No Hands: Aspects of Autonomous Vehicle Control
13 Arlette van Wissen (VU), Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains
14 Yangyang Shi (TUD), Language Models With Meta-information
15 Natalya Mogles (VU), Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare
16 Krystyna Milian (VU), Supporting trial recruitment and design by automatically interpreting eligibility criteria
17 Kathrin Dentler (VU), Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability
18 Mattijs Ghijsen (UVA), Methods and Models for the Design and Study of Dynamic Agent Organizations
19 Vinicius Ramos (TUE), Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support
20 Mena Habib (UT), Named Entity Extraction and Disambiguation for Informal Text: The Missing Link
21 Kassidy Clark (TUD), Negotiation and Monitoring in Open Environments
22 Marieke Peeters (UU), Personalized Educational Games - Developing agent-supported scenario-based training
23 Eleftherios Sidirourgos (UvA/CWI), Space Efficient Indexes for the Big Data Era
24 Davide Ceolin (VU), Trusting Semi-structured Web Data
25 Martijn Lappenschaar (RUN), New network models for the analysis of disease interaction
26 Tim Baarslag (TUD), What to Bid and When to Stop
27 Rui Jorge Almeida (EUR), Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty
28 Anna Chmielowiec (VU), Decentralized k-Clique Matching
29 Jaap Kabbedijk (UU), Variability in Multi-Tenant Enterprise Software
30 Peter de Cock (UvT), Anticipating Criminal Behaviour

31 Leo van Moergestel (UU), Agent Technology in Agile Multiparallel Manufacturing and Product Support

32 Naser Ayat (UvA), On Entity Resolution in Probabilistic Data

33 Tesfa Tegegne (RUN), Service Discovery in eHealth

34 Christina Manteli (VU), The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems.

35 Joost van Ooijen (UU), Cognitive Agents in Virtual Worlds: A Middleware Design Approach

36 Joos Buijs (TUE), Flexible Evolutionary Algorithms for Mining Structured Process Models

37 Maral Dadvar (UT), Experts and Machines United Against Cyberbullying

38 Danny Plass-Oude Bos (UT), Making brain-computer interfaces better: improving usability through post-processing.

39 Jasmina Maric (UvT), Web Communities, Immigration, and Social Capital

40 Walter Omona (RUN), A Framework for Knowledge Management Using ICT in Higher Education

41 Frederic Hogenboom (EUR), Automated Detection of Financial Events in News Text

42 Carsten Eijckhof (CWI/TUD), Contextual Multidimensional Relevance Models

43 Kevin Vlaanderen (UU), Supporting Process Improvement using Method Increments

44 Paulien Meesters (UvT), Intelligent Blauw. Met als ondertitel: Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden.

45 Birgit Schmitz (OUN), Mobile Games for Learning: A Pattern-Based Approach

46 Ke Tao (TUD), Social Web Data Analytics: Relevance, Redundancy, Diversity

47 Shangsong Liang (UVA), Fusion and Diversification in Information Retrieval

---

2015 01 Niels Netten (UvA), Machine Learning for Relevance of Information in Crisis Response

02 Faiza Bukhsh (UvT), Smart auditing: Innovative Compliance Checking in Customs Controls

03 Twan van Laarhoven (RUN), Machine learning for network data

04 Howard Spoelstra (OUN), Collaborations in Open Learning Environments

05 Christoph Bösch (UT), Cryptographically Enforced Search Pattern Hiding

06 Farideh Heidari (TUD), Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes

07 Maria-Hendrike Peetz (UvA), Time-Aware Online Reputation Analysis
08 Jie Jiang (TUD), Organizational Compliance: An agent-based model for designing and evaluating organizational interactions
09 Randy Klaassen (UT), HCI Perspectives on Behavior Change Support Systems
10 Henry Hermans (OUN), OpenU: design of an integrated system to support lifelong learning
11 Yongming Luo (TUE), Designing algorithms for big graph datasets: A study of computing bisimulation and joins
12 Julie M. Birkholz (VU), Modi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks
13 Giuseppe Procaccianti (VU), Energy-Efficient Software
14 Bart van Straalen (UT), A cognitive approach to modeling bad news conversations
15 Klaas Andries de Graaf (VU), Ontology-based Software Architecture Documentation
16 Changyun Wei (UT), Cognitive Coordination for Cooperative Multi-Robot Teamwork
17 André van Cleeff (UT), Physical and Digital Security Mechanisms: Properties, Combinations and Trade-offs
18 Holger Pirk (CWI), Waste Not, Want Not! - Managing Relational Data in Asymmetric Memories
19 Bernardo Tabuenca (OUN), Ubiquitous Technology for Lifelong Learners
20 Lois Vanhée (UU), Using Culture and Values to Support Flexible Coordination
21 Sibren Fetter (OUN), Using Peer-Support to Expand and Stabilize Online Learning
22 Zhemin Zhu (UT), Co-occurrence Rate Networks
23 Luit Gazendam (VU), Cataloguer Support in Cultural Heritage
24 Richard Berendsen (UVA), Finding People, Papers, and Posts: Vertical Search Algorithms and Evaluation
25 Steven Woudenberg (UU), Bayesian Tools for Early Disease Detection
26 Alexander Hogenboom (EUR), Sentiment Analysis of Text Guided by Semantics and Structure
27 Sándor Héman (CWI), Updating compressed colomn stores
28 Janet Bagorogoza (TiU), Knowledge Management and High Performance; The Uganda Financial Institutions Model for HPO
29 Hendrik Baier (UM), Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains

30 Kiavash Bahreini (OU), Real-time Multimodal Emotion Recognition in E-Learning
31 Yakup Koç (TUD), On the robustness of Power Grids
32 Jerome Gard (UL), Corporate Venture Management in SMEs
33 Frederik Schadd (TUD), Ontology Mapping with Auxiliary Resources
34 Victor de Graaf (UT), Gesocial Recommender Systems
35 Jungxao Xu (TUD), Affective Body Language of Humanoid Robots: Perception and Effects in Human Robot Interaction

---

2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VU), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UVA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VU), Measuring and modeling negative emotions for virtual training
08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VU), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UVA), Search Engines that Learn from Their Users
12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VU), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UVA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VU), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VU), Refining Statistical Data on the Web
19 Julia Efremova (Tu/e), Mining Social Structures from Genealogical Data
20 Daan Odijk (UVA), Context & Semantics in News & Web Search

21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
22 Grace Lewis (VU), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UVA), Query Auto Completion in Information Retrieval
24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
26 Dilhan Thilakarathne (VU), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
30 Ruud Mattheij (UvT), The Eyes Have It
31 Mohammad Khelghati (UT), Deep web content monitoring
32 Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
33 Peter Bloem (UVA), Single Sample Statistics, exercises in learning from just one example
34 Dennis Schunselaar (TUE), Configurable Process Trees: Elicitation, Analysis, and Enactment
35 Zhaochun Ren (UVA), Monitoring Social Media: Summarization, Classification and Recommendation
36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
40 Christian Detweiler (TUD), Accounting for Values in Design
41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance

42 Spyros Martzoukos (UVA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
44 Thibault Sellam (UVA), Automatic Assistants for Database Exploration
45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
46 Jorge Gallego Perez (UT), Robots to Make you Happy
47 Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
48 Tanja Buttler (TUD), Collecting Lessons Learned
49 Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
50 Yan Wang (UVT), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains

---

2017 01 Jan-Jaap Oerlemans (UL), Investigating Cybercrime
02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
05 Mahdieh Shadi (UVA), Collaboration Behavior
06 Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
08 Rob Konijn (VU) , Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
10 Robby van Delden (UT), (Steering) Interactive Play Behavior
11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
12 Sander Leemans (TUE), Robust Process Mining with Guarantees
13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
14 Shoshannah Tekofsky (UvT), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
15 Peter Berck (RUN), Memory-Based Text Correction

16 Aleksandr Chuklin (UVA), Understanding and Modeling Users of Modern Search Engines
17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
18 Ridho Reinanda (UVA), Entity Associations for Search
19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
22 Sara Magliacane (VU), Logics for causal inference under uncertainty
23 David Graus (UVA), Entities of Interest — Discovery in Digital Traces
24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
25 Veruska Zamborlini (VU), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
27 Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
28 John Klein (VU), Architecture Practices for Complex Contexts
29 Adel Alhuraibi (UvT), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
30 Wilma Latuny (UvT), The Power of Facial Expressions
31 Ben Ruijl (UL), Advances in computational methods for QFT calculations
32 Thaer Samar (RUN), Access to and Retrievability of Content in Web Archives
33 Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity
34 Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
35 Martine de Vos (VU), Interpreting natural science spreadsheets
36 Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging
37 Alejandro Montes Garcia (TUE), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
38 Alex Kayal (TUD), Normative Social Applications
39 Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR

	40	Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
	41	Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
	42	Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
	43	Maaike de Boer (RUN), Semantic Mapping in Video Retrieval
	44	Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering
	45	Bas Testerink (UU), Decentralized Runtime Norm Enforcement
	46	Jan Schneider (OU), Sensor-based Learning Support
	47	Jie Yang (TUD), Crowd Knowledge Creation Acceleration
	48	Angel Suarez (OU), Collaborative inquiry-based learning
2018	01	Han van der Aa (VUA), Comparing and Aligning Process Representations
	02	Felix Mannhardt (TUE), Multi-perspective Process Mining
	03	Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction
	04	Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks
	05	Hugo Huurdeman (UVA), Supporting the Complex Dynamics of the Information Seeking Process
	06	Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems
	07	Jieting Luo (UU), A formal account of opportunism in multi-agent systems
	08	Rick Smetsers (RUN), Advances in Model Learning for Software Systems
	09	Xu Xie (TUD), Data Assimilation in Discrete Event Simulations
	10	Julienka Mollee (VUA), Moving forward: supporting physical activity behavior change through intelligent technology
	11	Mahdi Sargolzaei (UVA), Enabling Framework for Service-oriented Collaborative Networks
	12	Xixi Lu (TUE), Using behavioral context in process mining
	13	Seyed Amin Tabatabaei (VUA), Computing a Sustainable Future
	14	Bart Joosten (UVT), Detecting Social Signals with Spatiotemporal Gabor Filters
	15	Naser Davarzani (UM), Biomarker discovery in heart failure

16 Jaebok Kim (UT), Automatic recognition of engagement and emotion in a group of children
17 Jianpeng Zhang (TUE), On Graph Sample Clustering
18 Henriette Nakad (UL), De Notaris en Private Rechtspraak
19 Minh Duc Pham (VUA), Emergent relational schemas for RDF
20 Manxia Liu (RUN), Time and Bayesian Networks
21 Aad Slootmaker (OUN), EMERGO: a generic platform for authoring and playing scenario-based serious games
22 Eric Fernandes de Mello Araujo (VUA), Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks
23 Kim Schouten (EUR), Semantics-driven Aspect-Based Sentiment Analysis
24 Jered Vroon (UT), Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots
25 Riste Gligorov (VUA), Serious Games in Audio-Visual Collections
26 Roelof Anne Jelle de Vries (UT),Theory-Based and Tailor-Made: Motivational Messages for Behavior Change Technology
27 Maikel Leemans (TUE), Hierarchical Process Mining for Scalable Software Analysis
28 Christian Willemse (UT), Social Touch Technologies: How they feel and how they make you feel
29 Yu Gu (UVT), Emotion Recognition from Mandarin Speech
30 Wouter Beek, The ”K” in ”semantic web” stands for ”knowledge”: scaling semantics to the web

---

2019 01 Rob van Eijk (UL),Web privacy measurement in real-time bidding systems. A graph-based approach to RTB system classification
02 Emmanuelle Beauxis Aussalet (CWI, UU), Statistics and Visualizations for Assessing Class Size Uncertainty
03 Eduardo Gonzalez Lopez de Murillas (TUE), Process Mining on Databases: Extracting Event Data from Real Life Data Sources
04 Ridho Rahmadi (RUN), Finding stable causal structures from clinical data
05 Sebastiaan van Zelst (TUE), Process Mining with Streaming Data
06 Chris Dijkshoorn (VU), Nichesourcing for Improving Access to Linked Cultural Heritage Datasets
07 Soude Fazeli (TUD), Recommender Systems in Social Learning Platforms
08 Frits de Nijs (TUD), Resource-constrained Multi-agent Markov Decision Processes
09 Fahimeh Alizadeh Moghaddam (UVA), Self-adaptation for energy efficiency in software systems

10 Qing Chuan Ye (EUR), Multi-objective Optimization Methods for Allocation and Prediction
11 Yue Zhao (TUD), Learning Analytics Technology to Understand Learner Behavioral Engagement in MOOCs
12 Jacqueline Heinerman (VU), Better Together
13 Guanliang Chen (TUD), MOOC Analytics: Learner Modeling and Content Generation
14 Daniel Davis (TUD), Large-Scale Learning Analytics: Modeling Learner Behavior & Improving Learning Outcomes in Massive Open Online Courses
15 Erwin Walraven (TUD), Planning under Uncertainty in Constrained and Partially Observable Environments
16 Guangming Li (TUE), Process Mining based on Object-Centric Behavioral Constraint (OCBC) Models
17 Ali Hurriyetoglu (RUN),Extracting actionable information from microtexts
18 Gerard Wagenaar (UU), Artefacts in Agile Team Communication
19 Vincent Koeman (TUD), Tools for Developing Cognitive Agents
20 Chide Groenouwe (UU), Fostering technically augmented human collective intelligence
21 Cong Liu (TUE), Software Data Analytics: Architectural Model Discovery and Design Pattern Detection
22 Martin van den Berg (VU),Improving IT Decisions with Enterprise Architecture
23 Qin Liu (TUD), Intelligent Control Systems: Learning, Interpreting, Verification
24 Anca Dumitrache (VU), Truth in Disagreement - Crowdsourcing Labeled Data for Natural Language Processing
25 Emiel van Miltenburg (VU), Pragmatic factors in (automatic) image description
26 Prince Singh (UT), An Integration Platform for Synchromodal Transport
27 Alessandra Antonaci (OUN), The Gamification Design Process applied to (Massive) Open Online Courses
28 Esther Kuindersma (UL), Cleared for take-off: Game-based learning to prepare airline pilots for critical situations
29 Daniel Formolo (VU), Using virtual agents for simulation and training of social skills in safety-critical circumstances
30 Vahid Yazdanpanah (UT), Multiagent Industrial Symbiosis Systems
31 Milan Jelisavcic (VU), Alive and Kicking: Baby Steps in Robotics
32 Chiara Sironi (UM), Monte-Carlo Tree Search for Artificial General Intelligence in Games

33 Anil Yaman (TUE), Evolution of Biologically Inspired Learning in Artificial Neural Networks

34 Negar Ahmadi (TUE), EEG Microstate and Functional Brain Network Features for Classification of Epilepsy and PNES

35 Lisa Facey-Shaw (OUN), Gamification with digital badges in learning programming

36 Kevin Ackermans (OUN), Designing Video-Enhanced Rubrics to Master Complex Skills

37 Jian Fang (TUD), Database Acceleration on FPGAs

38 Akos Kadar (OUN), Learning visually grounded and multilingual representations

---

2020 01 Armon Toubman (UL), Calculated Moves: Generating Air Combat Behaviour

02 Marcos de Paula Bueno (UL), Unraveling Temporal Processes using Probabilistic Graphical Models

03 Mostafa Deghani (UvA), Learning with Imperfect Supervision for Language Understanding

04 Maarten van Gompel (RUN), Context as Linguistic Bridges

05 Yulong Pei (TUE), On local and global structure mining

06 Preethu Rose Anish (UT), Stimulation Architectural Thinking during Requirements Elicitation - An Approach and Tool Support

07 Wim van der Vegt (OUN), Towards a software architecture for reusable game components

08 Ali Mirsoleimani (UL),Structured Parallel Programming for Monte Carlo Tree Search

09 Myriam Traub (UU), Measuring Tool Bias and Improving Data Quality for Digital Humanities Research

10 Alifah Syamsiyah (TUE), In-database Preprocessing for Process Mining

11 Sepideh Mesbah (TUD), Semantic-Enhanced Training Data Augmentation-Methods for Long-Tail Entity Recognition Models

12 Ward van Breda (VU), Predictive Modeling in E-Mental Health: Exploring Applicability in Personalised Depression Treatment

13 Marco Virgolin (CWI), Design and Application of Gene-pool Optimal Mixing Evolutionary Algorithms for Genetic Programming

14 Mark Raasveldt (CWI/UL), Integrating Analytics with Relational Databases

15 Konstantinos Georgiadis (OUN), Smart CAT: Machine Learning for Configurable Assessments in Serious Games

16 Ilona Wilmont (RUN), Cognitive Aspects of Conceptual Modelling

17 Daniele Di Mitri (OUN), The Multimodal Tutor: Adaptive Feedback from Multimodal Experiences
18 Georgios Methenitis (TUD), Agent Interactions & Mechanisms in Markets with Uncertainties: Electricity Markets in Renewable Energy Systems
19 Guido van Capelleveen (UT), Industrial Symbiosis Recommender Systems
20 Albert Hankel (VU), Embedding Green ICT Maturity in Organisations
21 Karine da Silva Miras de Araujo (VU), Where is the robot?: Life as it could be
22 Maryam Masoud Khamis (RUN), Understanding complex systems implementation through a modeling approach: the case of e-government in Zanzibar
23 Rianne Conijn (UT), The Keys to Writing: A writing analytics approach to studying writing processes using keystroke logging
24 Lenin da Nobrega Medeiros (VUA/RUN), How are you feeling, human? Towards emotionally supportive chatbots
25 Xin Du (TUE), The Uncertainty in Exceptional Model Mining
26 Krzysztof Leszek Sadowski (UU), GAMBIT: Genetic Algorithm for Model-Based mixed-Integer opTimization
27 Ekaterina Muravyeva (TUD), Personal data and informed consent in an educational context
28 Bibeg Limbu (TUD), Multimodal interaction for deliberate practice: Training complex skills with augmented reality
29 Ioan Gabriel Bucur (RUN), Being Bayesian about Causal Inference
30 Bob Zadok Blok (UL), Creatief, Creatieve, Creatiefst
31 Gongjin Lan (VU), Learning better – From Baby to Better
32 Jason Rhuggenaath (TUE), Revenue management in online markets: pricing and online advertising
33 Rick Gilsing (TUE), Supporting service-dominant business model evaluation in the context of business model innovation
34 Anna Bon (MU), Intervention or Collaboration? Redesigning Information and Communication Technologies for Development
35 Siamak Farshidi (UU), Multi-Criteria Decision-Making in Software Production

---

2021 01 Francisco Xavier Dos Santos Fonseca (TUD),Location-based Games for Social Interaction in Public Space
02 Rijk Mercuur (TUD), Simulating Human Routines:Integrating Social Practice Theory in Agent-Based Models
03 Seyyed Hadi Hashemi (UVA), Modeling Users Interacting with Smart Devices

04 Ioana Jivet (OU), The Dashboard That Loved Me: Designing adaptive learning analytics for self-regulated learning
05 Davide Dell'Anna (UU), Data-Driven Supervision of Autonomous Systems
06 Daniel Davison (UT), "Hey robot, what do you think?" How children learn with a social robot
07 Armel Lefebvre (UU), Research data management for open science
08 Nardie Fanchamps (OU), The Influence of Sense-Reason-Act Programming on Computational Thinking
09 Cristina Zaga (UT), The Design of Robothings. Non-Anthropomorphic and Non-Verbal Robots to Promote Children s Collaboration Through Play
10 Quinten Meertens (UvA), Misclassification Bias in Statistical Learning
11 Anne van Rossum (UL), Nonparametric Bayesian Methods in Robotic Vision
12 Lei Pi (UL), External Knowledge Absorption in Chinese SMEs
13 Bob R. Schadenberg (UT), Robots for Autistic Children: Understanding and Facilitating Predictability for Engagement in Learning
14 Negin Samaeemofrad (UL), Business Incubators: The Impact of Their Support
15 Onat Ege Adali (TU/e), Transformation of Value Propositions into Resource Re-Configurations through the Business Services Paradigm
16 Esam A. H. Ghaleb (UM), BIMODAL EMOTION RECOGNITION FROM AUDIO-VISUAL CUES
17 Dario Dotti (UM), Human Behavior Understanding from motion and bodily cues using deep neural networks
18 Remi Wieten (UU), Bridging the Gap Between Informal Sense-Making Tools and Formal Systems - Facilitating the Construction of Bayesian Networks and Argumentation Frameworks
19 Roberto Verdecchia (VU), Architectural Technical Debt: Identification and Management
20 Masoud Mansoury (TU/e), Understanding and Mitigating Multi-Sided Exposure Bias in Recommender Systems